DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM AND ZX SPECTRUM+

GRAPHICS

PIERS LETCHER

BOOK FOUR

Advanced sprite programming techniques plus a full-colour design directory

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM ZX SPECTRUM+

GRAPHICS

THE DK SCREEN-SHOT PROGRAMMING SERIES

Books One and Two in the DK Screen-Shot Programming Series brought to home computer users a new and exciting way of learning how to program in BASIC. Following the success of this completely new concept in teach-yourself computing, the series now carries on to explore the speed and potential of machine-code graphics. Fully illustrated in the unique Screen-Shot style, the series continues to set new standards in the world of computer books.

BOOKS ABOUT THE ZX SPECTRUM+

This is Book Four in a series of guides to programming the ZX Spectrum+. It contains a complete machine-code sprite-programming course for the Spectrum+, and features its own sprite editor which enables you to design and store sprites directly from the keyboard. Together with its companion volumes, it builds up into a complete programming and graphics system.

ALSO AVAILABLE IN THE SERIES

Step-by-Step Programming for the **Commodore 64**

Step-by-Step Programming for the **BBC Micro**

Step-by-Step Programming for the **Acorn Electron**

Step-by-Step Programming for the **Apple IIe**

Step-by-Step Programming for the **Apple IIc**

PIERS LETCHER

After graduating with a degree in Computer Systems, Piers Letcher has worked in many areas of the computer industry, from programming and selling mainframes to designing and marketing educational software. He was Peripherals Editor of *Personal Computer News* until May 1984 and has since written a guide to peripherals and a number of other books for popular home micros.

BOOK FOUR


sinclair
ZX Spectrum +

Screen Shot
PROGRAMMING SERIES

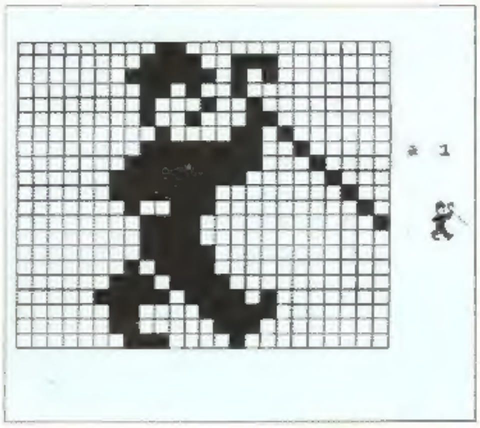
STEP-BY-STEP PROGRAMMING
ZX SPECTRUM
ZX SPECTRUM +
GRAPHICS

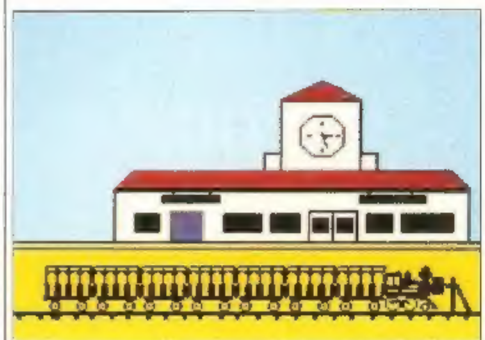
PIERS LETCHER

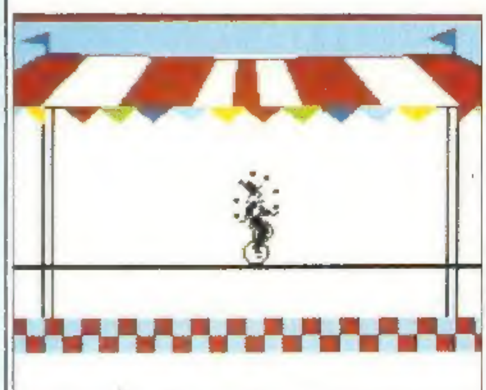
DORLING KINDERSLEY · LONDON

BOOK FOUR

CONTENTS







The DK Screen-Shot Programming Series was conceived, edited and designed by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

Editor Michael Upshall
Designer Steve Wilson
Photographer Vincent Oliver
Series Editor David Burnie
Series Art Editor Peter Luff
Managing Editor Alan Buckingham

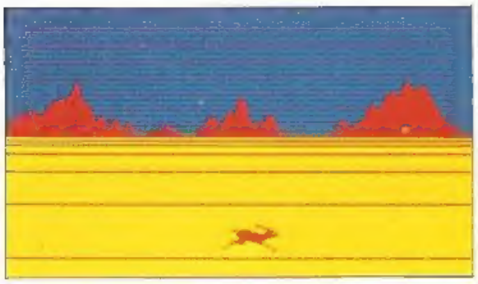
First published in Great Britain in 1985 by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.
Second impression 1985

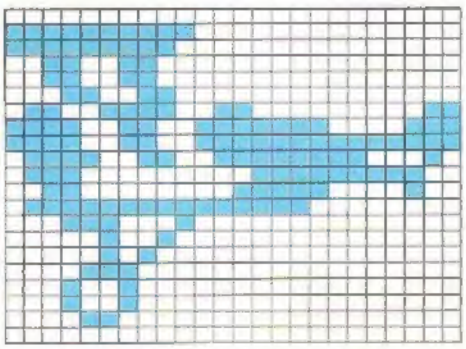
British Library Cataloguing in Publication Data

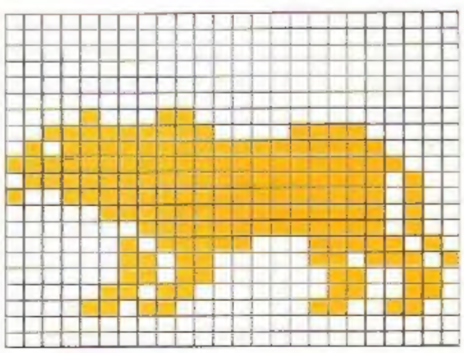
Letcher, Piers
Step-by-step programming ZX Spectrum and ZX Spectrum+ Graphics.
– (DK screen shot programming series) Bk. 4
1. Sinclair ZX Spectrum (Computer) – Programming
I. Title
001.64'2 QA76.8.S625

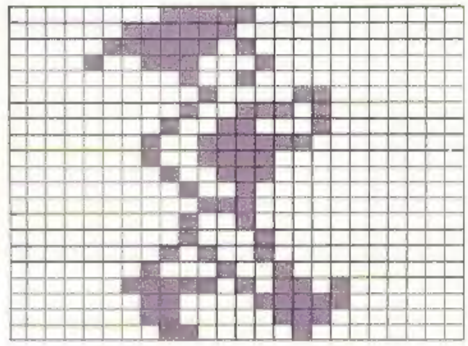
ISBN 0-86318-104-X

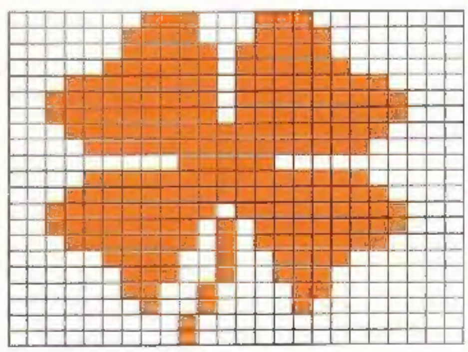
Typesetting by Gedset Limited, Cheltenham, England
Reproduction by Reprocolor Llovet S.A., Barcelona, Spain and F. E. Burman Limited, London
Printed and bound in Italy by A. Mondadori, Verona











ABOUT THIS BOOK

The Sinclair Spectrum is one of the most popular microcomputers ever produced. One reason for its success has been its remarkable ability to produce graphic displays rivalling those produced by much larger computers designed only ten or fifteen years ago. However, graphics programming in BASIC under-utilizes the Spectrum. To produce the kind of displays seen in commercially available games, you need to use machine code as well as BASIC.

What is machine code?

The heart of the Spectrum, the Z80 central processor, cannot understand BASIC. A BASIC program must first be translated into a simpler language that the machine can understand (hence the term "machine code"). This code is in the form of binary 1s and 0s. Before the processor can execute a BASIC program line, all keywords and variables are first converted to machine-code instructions.

BASIC is an example of what is known as an "interpreted", as opposed to a "compiled", language — that is, it is executed by the central processor line by line rather than as a complete program. While an interpreted language is easier to use, it is also slower in execution. By writing programs in machine code, you can miss out the BASIC interpreter altogether. In addition, machine code allows you to utilize many features of your Spectrum which cannot be reached from BASIC, so that you can therefore achieve far more impressive results than would ever be possible from the simpler, but more restricted, BASIC. You can get an idea of how much faster machine code is by seeing the time taken for the programs in this book to run.

Disadvantages of machine code

Given all the advantages of machine code in both speed and flexibility, why not ignore BASIC and use machine code all the time? The answer is simply convenience. Using machine code is time-consuming, difficult and frustrating, and attempting to write your own code is only for the expert. When you see machine-code listings, they are usually in a "disassembled" form, that is, with some of the numbers translated into mnemonics such as LD for LOAD, and JP for JUMP. But a special disassembler program is required simply to give you a machine-code listing in this form, and these mnemonics are themselves far from simple. Using machine code even the simplest operations in BASIC, such as drawing a line on the screen, require many lines of programming. In addition, machine code has no error-trapping routines such as those in BASIC. If a mistake is made when keying in a BASIC program, the program will not be lost (although the program may refuse to RUN at some point); in machine code, without error-trapping routines, a mistake will probably cause the Spectrum to crash, with the result that both the program and its DATA are lost.

The solution

This book combines the advantages of machine code with the convenience and simplicity of BASIC. This is done by giving the machine code in the form of ready-made and tested routines, which you can then use in your BASIC programs. The machine code is shown as DATA statements in BASIC, which means it isn't necessary for you to understand anything about machine code to be able to use the routines. The DATA is given in the form of decimal numbers, rather than in binary or hexadecimal (to base 16), so that the machine code is in the form most convenient for you to key in.

The machine-code routines

The screen below shows an example of a machine code routine (the double vertical sprite routine, FNj, given on page 17).

DOUBLE VERTICAL SPRITE ROUTINE

```
7450 LET b=52100: LET l=225: LET
 z=0: RESTORE 7460
7451 FOR i=0 TO l-1: READ a
7452 POKE (b+i),a: LET z=z+a
7453 NEXT i
7454 LET z=INT (((z/l)-INT (z/l)
)*l)
7455 READ a: IF a<>z THEN PRINT
"??": STOP

7460 DATA 0,42,11,92,17
7461 DATA 4,0,25,78,30
7462 DATA 6,25,70,25,126
7463 DATA 230,3,50,103,204
7464 DATA 25,126,50,104,204
7465 DATA 25,126,230,1,50
7466 DATA 105,204,25,126,230
7467 DATA 1,50,101,204,25
7468 DATA 126,17,63,0,33
7469 DATA 9,213,25,61,32

7470 DATA 252,175,50,102,204
7471 DATA 205,93,210,205,86
```

Each routine in the book is shown like this, in the form of a BASIC program. The machine code is contained as a series of DATA statements in lines 7460 onwards. At the beginning of the routine, in lines 7450 to 7455, there are a few lines of BASIC. This is a loader program; variable b tells the computer where in memory to begin loading the routine, and variable l the number of bytes in the routine. When the loader routine is RUN, this routine is placed in memory from address 52100 onwards, and has a total length of 225 bytes.

As shown here, of course, the routine is simply a list of numbers, and has no visible meaning. These numbers are the ready-tested and assembled machine code which has then been converted to a sequence of decimal numbers. Each number corresponds to a single instruction or item of DATA required by the routine;

hence, all the numbers have values between 0 and 255, the maximum range of a byte. All you need to know about the routine is what it does and what information it requires so that you can call it correctly from your BASIC program.

All the routines in the book are defined as functions. Each function is individually coded by the letters a to o; a complete list of functions is given on pages 62-63. Demonstration BASIC programs can be found on the same page as each routine; these give you an indication of the kind of displays which are possible using the machine code.

How to use the routines

To use any program in this book, simply key in a machine-code routine together with a BASIC program which demonstrates its use. You will find full details of how to do this on pages 8-9. When you RUN the program, you will begin to see the true power of your Spectrum.

As you progress through the book and the range of routines grows, the BASIC programs grow too by calling several routines to produce increasingly complex displays. By keying in each routine, and then SAVEing it onto cassette or Microdrive, you will have a sophisticated but flexible graphics capability at your fingertips.

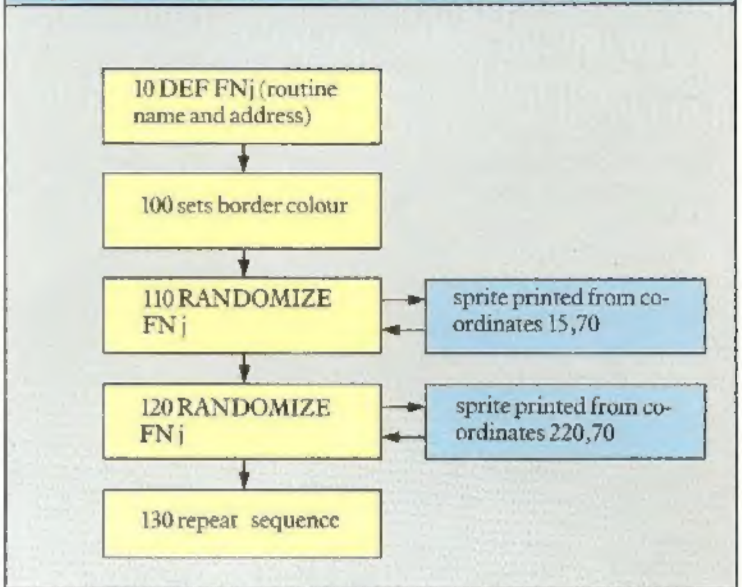
The programs in use

A typical program from this book (the unicycle program on page 23) contains two details which will be unfamiliar to BASIC programmers who have not used machine code before:

UNICYCLE PROGRAM

First, you will see in line 10 a DEF FN statement, which is used to instruct the computer that a machine-code routine with two parameters (x and y) is located at address 52100 in memory. You will also notice two RANDOMIZE FN commands (lines 110 and 120). These are the calls to the double vertical sprite routine, and the numbers in brackets which follow them are the parameter values to be passed to the machine-code routine (in this case, the start co-ordinates of the sprite, its direction, how far it is to move and various other instructions). When RUNning, the program is carried out by the computer in this way:

HOW THE UNICYCLE PROGRAM WORKS



On the left side of this diagram is the main BASIC program, and on the right you can see the machine-code routines, called twice using a RANDOMIZE FN statement. You will see from the diagram that the machine-code is used here very much as a subroutine would be used in BASIC, with variables passed to the routines each time they are called.

What the routines do

The routines in this book free you from the limitations of programming in BASIC. By using the machine-code given here, you will be able to create and control sprites, to control them on the screen and to animate them, and to scroll both the entire screen and defined areas of it.

In addition, two of the later routines provide an introduction to one of the most exciting aspects of machine-code graphics: interrupt-driven routines, which operate independently of BASIC, and which enable you to program your Spectrum to carry out several tasks simultaneously.

Creating and editing sprites

To make sprites even easier to use, a directory of over 200 sprites is included from pages 36 to 61. These sprites can be keyed in and then edited with the sprite editor routine and program, given on pages 11 to 13. Using the sprite editor, you will find it easy to make your own versions of the sprites given in this book. Using single-key commands, for example, you can invert the sprites, make them face another direction, or turn them upside down.

USING THE MACHINE CODE

The machine-code routines in this book can easily be incorporated into your BASIC programs without you having to understand the intricacies of how they work. Simply choose a program from this book, and follow the steps given here.

1: CLEAR memory

As soon as you switch on your Spectrum, type CLEAR 49000. This command resets RAMTOP, the top of the area in memory free for BASIC programs, and ensures that BASIC programs cannot overlap with the machine code stored in memory from 49200 upwards. Now you can safely use NEW to delete BASIC programs without losing any of the machine code in memory.

Remember to use CLEAR before loading machine code, since this command erases whatever is in memory above the specfied address.

2: Load the machine code

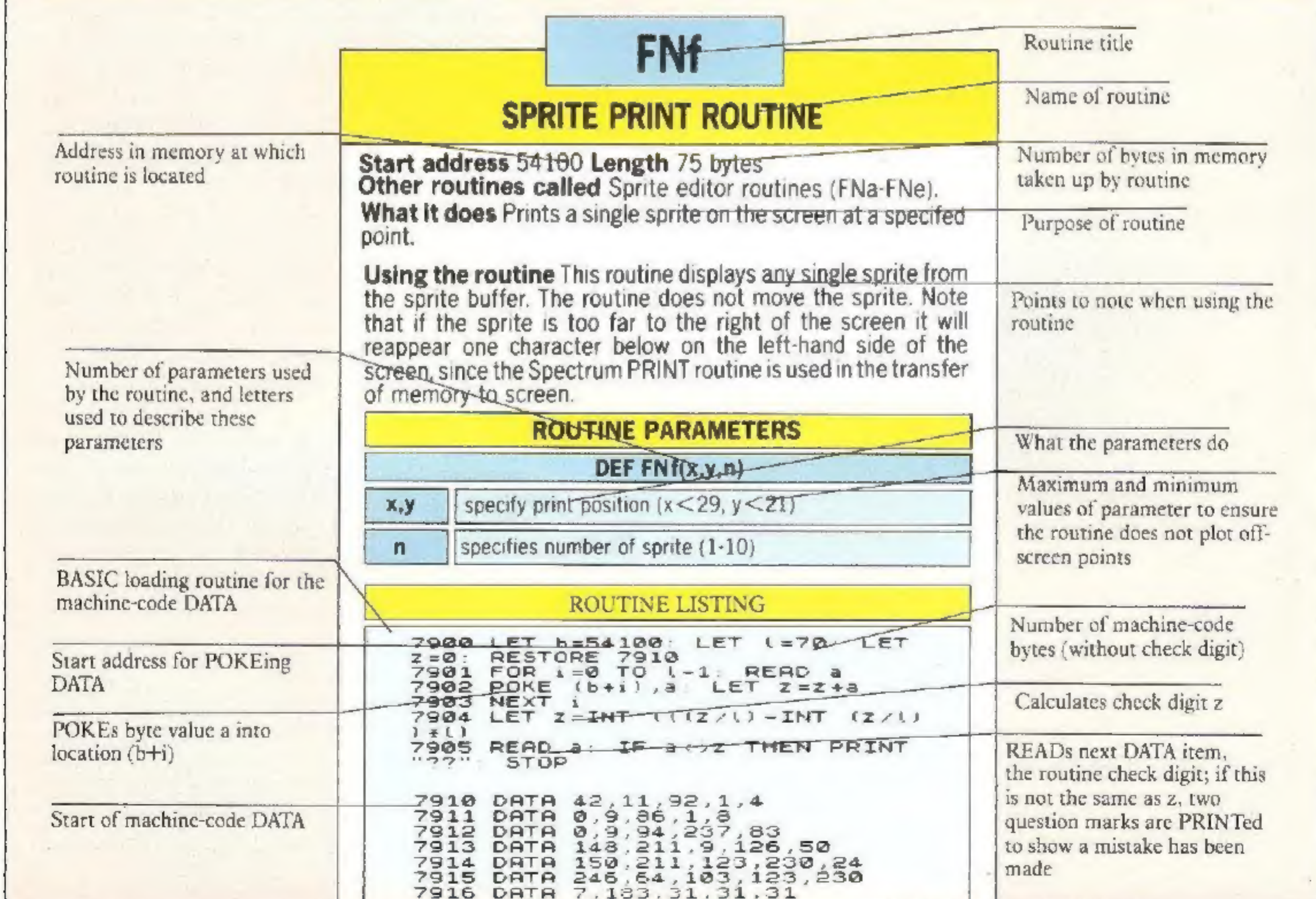
Now type in whatever machine-code routines are required by the BASIC program. After keying in the routine, RUN the short BASIC program which accompanies it; this loads the code into memory. If you keyed in the DATA correctly, you will see an "OK" message on the screen; if not, you will see a couple of question marks. In this case, look again at what you have typed in to trace the mistake.

3: SAVE the routine

When you are sure you have keyed in the routine correctly, SAVE it onto cassette or Microdrive. Always SAVE machine code before using it, to minimize the risk of losing everything you have keyed in. When BASIC errors occur, an error message is usually produced but the program is not lost. Machine-code routines, however, do not generally have error-trapping facilities, and a fault in the code will as often as not cause the Spectrum to crash — deleting everything in memory.

The machine code can be SAVEd in two ways: either in the form of DATA statements like any other BASIC

EXPLANATION OF A MACHINE-CODE BOX



listing, or, after you have loaded it into memory, as a block of code. To save machine code, type:

SAVE "routine name" CODE start address, length in bytes

The start address and length are given at the top of each machine-code box. The diagram on the facing page shows how this information is displayed.

4: LOAD a BASIC program

With the machine-code routine in memory, you can now use it in a BASIC program. DEF FN statements are used to tell the Spectrum the whereabouts of the routine in memory, and what information it requires.

Using functions

A machine-code routine can be called simply by specifying its start location, like this:

```
10 RANDOMIZE USR 54100
```

A line like this in a BASIC program, however, is not very informative. It tells you neither what the routine does, nor how many parameters the routine may require when called. This information could be POKEd into the appropriate memory locations – but the consequences of a mistake could be disastrous. Much more reliable is to pass information to the routines using a BASIC function. Functions on the Spectrum are identified by a single letter, and are followed by parameters in brackets. When you define the name and location of the function in your program, you must also specify the parameters, if any, which are to be passed to the routine. For example, the sprite print routine, FNf, requires three parameters:

```
10 DEF FN f(x,y,n)=USR 54100
```

Which letters are used after DEF FN is not important; their function is only to tell the computer the number of parameters which will follow the routine call in a BASIC program.

A machine-code function can be called from BASIC in two main ways, both of which require you to combine the keywords FN or USR with a BASIC keyword. The method used generally in this book is with the keyword RANDOMIZE. Thus,

```
20 RANDOMIZE FN f(10,10,1)
```

would display the first sprite from the sprite buffer in memory at co-ordinates 10,10. Note that using RANDOMIZE also resets the random number generator with a new seed; this may cause problems if you are also using a random function in your program. The second word you can use to call machine code is RESTORE. However, RESTORE also resets the pointer to DATA statements when you use it — which is of course the purpose of the RESTORE statement. If you opt to use RESTORE instead of RANDOMIZE then be especially careful if there are any READ or DATA statements in your program.

QUESTIONS AND ANSWERS

What if I make a mistake in keying in?
Don't panic! Nobody keys in long lists of numbers without making any mistakes. A check routine is included with each machine-code routine to warn you if you made any mistakes in keying in the DATA. This routine compares the DATA you have entered with a check number, which is placed by itself on the last DATA line of each routine.

After the loading program has POKEd the DATA numbers into memory, it looks to see if the check digit is the same as the one currently calculated. If the two numbers are different, the program prints two question marks to show an error has been made. If this happens, look through the numbers you have typed in to find the mistake. Having corrected the error, you may still find that the routine fails to load correctly; look to see if you have made more than one error.

Can I start anywhere in the book?
Yes, you can start on any page, but obviously when you key in a program it will not RUN unless the machine-code routine it calls is present in memory. Check before you begin if the program you want to RUN calls any machine-code routines you haven't already keyed in.
If you key in all the routines in this book as BASIC DATA statements, you will find there isn't room in memory to store them all. By loading each routine as machine code as soon as you have keyed it in, you can avoid this happening. In the form of machine code, you can, of course, use any of the Book Four routines together, as well as any routines from Book Three – the routines will not overlap in memory.

Can I adapt the BASIC programs?
Yes. You can edit the BASIC programs in any way you want to produce different displays, and you will find suggestions for variations throughout the book. One suggestion, though, if you are going to experiment with unusual or off-screen values for the machine-code parameters, is to SAVE what you have keyed in before experimenting. This will prevent you from losing hours of work at the keyboard!

Can I adapt the machine-code routines?
Yes, but at your own risk! Without a good understanding of machine code, it is highly unlikely that you will be able to alter any of the routines successfully. Much more probable is that the Spectrum would crash, with the result that both program and code are wiped from memory.

WHAT ARE SPRITES?

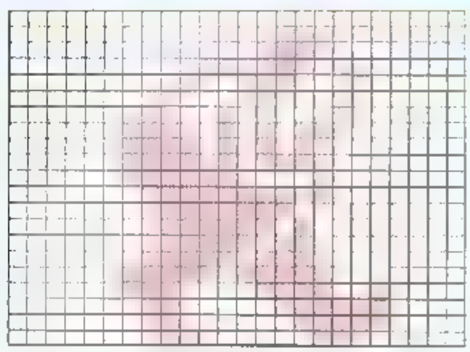
Most of the computer graphics you have created up to now have probably been stationary rather than moving, though you will have seen all sorts of moving graphics in arcade games and in commercial Spectrum software. Before you can create moving graphics for yourself, however, it is necessary to look at the ideas behind movement.

What is movement?

You tell something is moving if its position changes relative to something else. You know a train going past a window is moving, because the window is still. This book is about creating movement, and displaying moving objects. The objects to be moved are called sprites: objects which can move over a background without destroying it. The diagram below shows a single sprite..

EXAMPLE OF A SPRITE

21



24

Creating movement

The problem in creating movement is not so much in making something move as in making it move smoothly. You probably know that one way of getting something to move in BASIC on your Spectrum is to PRINT an object on the screen, wipe it off again, and PRINT it again in quick succession. This method has several disadvantages, the most important of which you will notice as soon as you try it out — the movement looks jerky. This is because there will be a space of one character between each position where the object is PRINTed. The other problem with BASIC is that it is simply not fast enough in operation to be used for smooth movement.

The jump of eight pixels between each position of the object is easily visible, and the obvious solution is to print the object every pixel rather than every character. This is easier said than done, however, since printing objects across pixel boundaries requires the step from BASIC to machine code. Using machine code will also give you the increase in speed which is necessary for implementing smooth movement.

To use movement effectively you must first make a few decisions on what you plan to move around the screen. Firstly, you must decide the size of object you want to move. The most obvious choice is a single character (8 by 8 pixels), but this looks very small in screen displays. On the other hand if you pick a size which is too large you will have the problem of trying to move thousands of pixels at the same time, resulting in a very jerky effect. The solution presented here is to use a shape 24 pixels wide by 21 pixels deep — or in character terms a little under three by three characters. To create a practical illusion of movement you must also make sure that the object to be moved does not destroy the background over which it passes.

Ways of implementing sprites

Some computers have sprites built into them as part of the machine hardware. The Commodore 64, for example, has a sophisticated chip dedicated to the implementation of sprites, since the method used to display them is so complicated. The chip works by saving the area of the screen underneath the sprite position (a block 24 by 21 pixels), printing the sprite, wiping off the sprite, printing the sprite one pixel further on and then replacing that part of the background which was uncovered by the movement. Though it would be possible to implement this process on the Spectrum, it would be very slow, or alternatively it would require a very long routine with many hundreds of bytes of DATA to be typed in.

An alternative method of implementing sprites uses a technique that you have probably used quite a lot already, which is to print onto the screen using Exclusive/OR plotting. If you do this with sprites you do not have to worry about the background, as it will remain unchanged, with the sprite appearing to move across the background without interference.

SPRITE SCREEN DISPLAY

THE SPRITE EDITOR 1

In order to use sprites, you need some means of creating them, and you need a location in memory where a number of them can be stored for future reference. This is the purpose of the sprite editor program. The program allows you to design and edit sprites on the screen, and stores them in memory for use by the sprite routines. Each sprite consists of 504 pixels, and is stored in 63 bytes of the Spectrum memory.

The sprite editor allows ten sprites to be defined at a time, and gives you the option of transferring sprites from one location in the sprite buffer to another. In addition, sprites can be flipped horizontally and vertically, and inverted (by switching the ink and paper attributes). The program also allows you to load in previously edited sprites from tape, and to save the current batch to tape for future use.

The sprite editor is a combination of BASIC and machine code. The code comprises six different routines, all linked together. The purpose of each routine is explained below.

The sprite editor routines

Routine FNa, at address 54200, is the base routine for the editor. It converts the large grid on the screen to the small display of the current sprite you see on the right. The current sprite is temporarily stored in a buffer, and the routine converts each dot in the buffer to a square on the screen.

When you want to save the current sprite, you have to decide which of the ten sprite positions you are saving to. A routine at 54317 transfers the sprite stored in the buffer to the appropriate location in the sprite table in memory. Routine FNb, at address 54353, carries out the reverse of this operation, transferring a sprite from its position in the sprite table to the sprite editor buffer. Routine FNa is then called again to display the sprite.

SPRITE EDITOR PROGRAM

```
  10 DEF FN a()=USR 54200
  20 DEF FN b()=USR 54353
  30 DEF FN c()=USR 54422
  40 DEF FN d()=USR 54436
  50 DEF FN e()=USR 54482
 100 LET a=22529: LET j=0: LET s
=a: LET o=s
 110 GO SUB 1200
 120 GO TO 560
 130 POKE o,233
 140 LET g$=INKEY$
 150 RANDOMIZE FN a()
 160 PAUSE 1.5
 170 LET o=s: LET j=PEEK o
 180 IF g$="I" THEN RANDOMIZE FN
c(): RANDOMIZE FN a()
 190 IF g$="P" THEN RANDOMIZE FN
d(): RANDOMIZE FN a()
 200 IF g$="U" THEN RANDOMIZE FN
e(): RANDOMIZE FN a()
 210 IF g$="Z" THEN GO TO 480
 220 IF g$<"4" OR g$>"8" THEN GO

scroll?
```

```
TO 240
 230 GO TO 370
 240 IF g$<"%" OR g$>"(" THEN GO
TO 140
 250 IF g$="%" THEN LET s=s-1
 260 IF g$="&" THEN LET s=s+32
 270 IF g$="'" THEN LET s=s-32
 280 IF g$="(" THEN LET s=s+1
 290 GO SUB 1160
 300 IF l=0 THEN LET s=s+1: GO T
O 140
 310 IF l>24 THEN LET s=s-1: GO
TO 140
 320 IF s<a THEN LET s=s+32: GO
TO 140
 330 IF s>a-1+21*32 THEN LET s=s
-32: GO TO 140
 340 POKE o,0
 350 POKE s,130
 360 GO TO 140
 370 IF g$="5" THEN LET s=s-1
 380 IF g$="6" THEN LET s=s+32

scroll?
```

SPRITE EDITOR PROGRAM CONTD.

```
 390 IF g$="7" THEN LET s=s-32
 400 IF g$="8" THEN LET s=s+1
 410 GO SUB 1160
 420 IF l=0 THEN LET s=s+1: GO T
O 140
 430 IF l>24 THEN LET s=s-1: GO
TO 140
 440 IF s<a THEN LET s=s+32
 450 IF s>a-1+21*32 THEN LET s=s
-32
 460 POKE o,56
 470 POKE s,135: GO TO 140
 480 PRINT AT 1,26;"SAVE "
 490 PRINT AT 3,26;"a-j"
 500 LET g$=INKEY$
 510 IF g$<"a" OR g$>"j" THEN GO
TO 500
 520 LET v=CODE g$-96: POKE 2330
0,v
 530 RANDOMIZE USR 54317
 540 BEEP .4,20
 550 GO SUB 1000

scroll?
```

```
 560 PRINT AT 1,26;"E a-j"
 570 PRINT AT 3,26;"1 SAVE"
 580 PRINT AT 5,26;"2 LOAD"
 590 LET g$=INKEY$
 600 IF g$="1" THEN GO TO 1100
 610 IF g$="2" THEN GO TO 1130
 620 IF g$<"a" OR g$>"j" THEN GO
TO 590
 630 LET v=CODE g$-96: POKE 2330
0,v
 640 GO SUB 1000
 650 RANDOMIZE FN b()
 660 PRINT AT 7,28;g$;" ";v;" "
 670 BEEP .4,30
 680 GO TO 130
1000 PRINT AT 1,26;"      "
1010 PRINT AT 3,26;"      "
1020 PRINT AT 5,26;"      "
1030 RETURN
1100 INPUT "SAVE ";n$: IF n$=""
THEN GO TO 1100
1110 SAVE n$CODE 54600,699

scroll?
```

THE SPRITE EDITOR 2

The first two routines in the sprite editor enabled you to draw, load and save sprites. The remaining sprite routines have been written to give you the means of manipulating the sprite you have drawn. The effect they have on the sprite is shown in the displays here. Each routine is called by a keypress. Thus, when CAPS SHIFT and I are pressed together, a routine is called to invert every pixel of the sprite. To change the direction in which the sprite is facing, routine FNe at address 54436 is used, called by pressing CAPS SHIFT and R. Finally, to turn the current sprite upside down, routine FNf is used (called by pressing CAPS SHIFT and U).

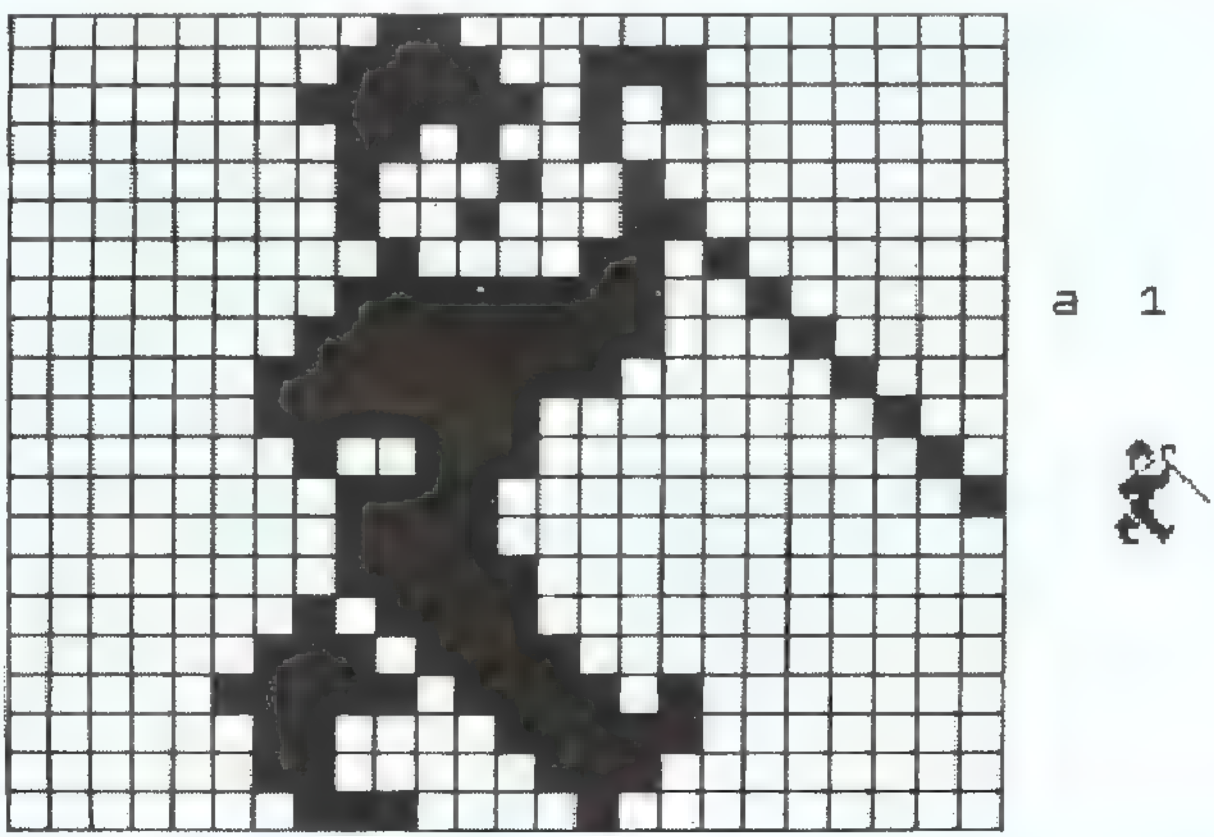
SPRITE EDITOR PROGRAM CONTD.

```
1120 GO TO 40
1130 INPUT "LOAD ";n$: LOAD n$ C
ODE : CLS
1140 GO TO 40
1150 STOP
1160 LET h=INT (s/32): LET l=s-h
*32
1170 RETURN
1200 FOR i=8 TO 200 STEP 8
1210 PLOT i,8: DRAW 0,167
1220 NEXT i
1230 FOR i=8 TO 175 STEP 8
1240 PLOT 8,i: DRAW 192,0
1250 NEXT i
1260 PLOT 8,175: DRAW 192,0
1270 RETURN


0 OK, 0:1
```

The BASIC editor program

The BASIC controlling program (shown on this page and on page 11) works as follows. Line 110 calls a subroutine at line 1200 to draw the grid on the screen (a simple series of lines). Line 120 calls a section of the program which prints the menu to the right of the grid. From this section blocks of the program at lines 1000, 1100 and 1130 are called as required. Lines 1000-1020 wipe the menu off the right-hand side of the screen. Lines 1100-1110 save the current sprite table to tape, and return control to the start of the program. A BEEP is heard whenever a sprite is saved in memory. Lines 1130-1140 load a new sprite table from tape into memory. If neither 1100 or 1130 is called, control returns to line 130.

Line 150 prints a sprite on the screen. Lines 180-210 accept a keypress and check if it is one of the functions I, R or U. If so, the relevant machine-code routine is called. If



FNa-e

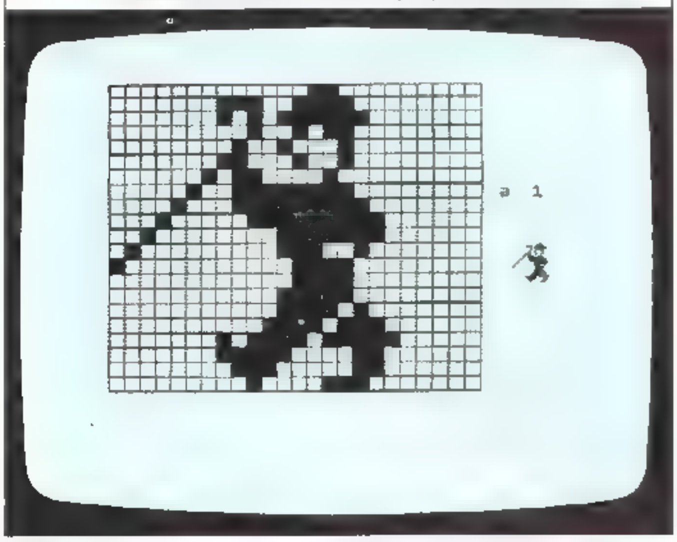
24 x 21 SPRITE EDITOR ROUTINES

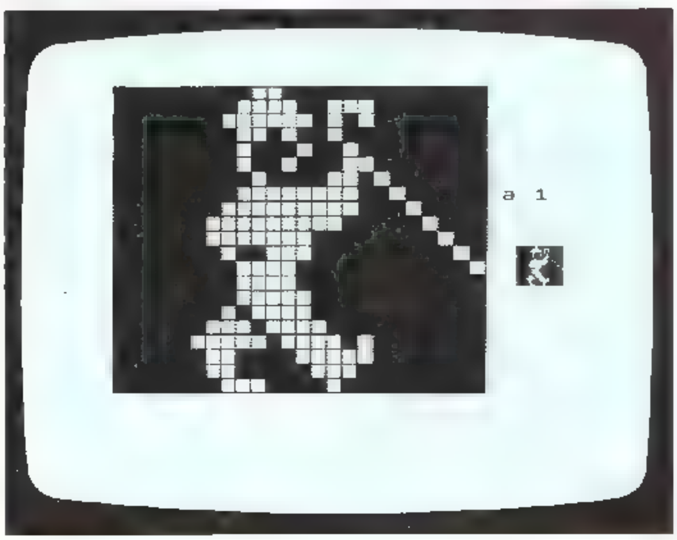
Start addresses 54200,54353,54422,54436,54482
Length 355 bytes
What they do The routines allow the user to design and edit up to ten sprites, and save them for use by later routines.

Using the routines The routines work in conjunction with a BASIC program shown on page 11. Editing the current sprite is controlled by the 5, 6, 7 and 8 keys; used with the shift key, pixels are inked in; used without, pixels are unset or, if set, changed to paper. The current sprite can be inverted (CAPS SHIFT and I), turned to face the other direction horizontally (CAPS SHIFT and R), or turned upside down (CAPS SHIFT and U).
Sprites are stored in a table from location 54600. Previously defined sprites may be loaded from tape, altered, and then saved as a new selection. When you save the sprite table, remember that all ten sprites are saved, not just the one you have most recently been editing.

the keypress is CAPS SHIFT and Z, control goes to the short sequence at 480 to display the menu again. Lines 250 to 280 do the work of editing the sprite. The subroutine at line 1160 is called from this section. Notice how line 1160 divides the address to be saved into two parts, since 16 bit addresses will not fit into a single byte.

MANIPULATING THE CURRENT SPRITE





ROUTINE LISTING

```
7950 LET b=54200: LET l=350: LET
 z=0: RESTORE 7960
7951 FOR i=0 TO l-1: READ a
7952 POKE (b+i),a: LET z=z+a
7953 NEXT i
7954 LET z=INT (((z/l)-INT (z/l)
)*l)
7955 READ a: IF a<>z THEN PRINT
"??": STOP

7960 DATA 33,250,212,17,1
7961 DATA 88,14,3,213,237
7962 DATA 83,246,212,6,21
7963 DATA 237,91,246,212,197
7964 DATA 6,8,26,254,0
7965 DATA 40,4,254,130,32
7966 DATA 6,55,203,22,195
7967 DATA 224,211,167,203,22
7968 DATA 19,16,235,229,42
7969 DATA 246,212,1,32,0

7970 DATA 9,34,246,212,225
7971 DATA 193,35,16,212,209
7972 DATA 62,8,131,95,13
7973 DATA 32,197,33,250,212
7974 DATA 17,123,72,6,3
7975 DATA 213,197,14,3,6
7976 DATA 8,213,126,18,20
7977 DATA 35,16,250,209,62
7978 DATA 32,131,9[illegible],121,61
7979 DATA 254,1,32,5,6

7980 DATA 5,79,24,233,6
7981 DATA 8,79,254,0,32
7982 DATA 226,193,209,19,16
7983 DATA 215,2[illegible]1,205,64,212
7984 DATA 34,248,212,237,91
7985 DATA 248,212,33,250,212
7986 DATA 1,63,0,237,176
7987 DATA 201,58,4,91,71
7988 DATA 33,72,213,17,63
7989 DATA 0,167,237,[illegible]2,25

7990 DATA 16,253,201,205,64
7991 DATA 212,34,248,212,17
7992 DATA 1,88,14,3,213
7993 DATA 237,83,246,212,6
7994 DATA 21,237,91,246,212
7995 DATA 197,6,8,126,167
7996 DATA 203,39,245,48,5
7997 DATA 62,0,195,120,212
7998 DATA 62,56,18,241,19
7999 DATA 16,239,229,42,246

8000 DATA 212,1,32,0,9
8001 DATA 34,246,212,225,193
8002 DATA 35,16,214,209,62
8003 DATA 8,131,95,13,32
8004 DATA 199,201,33,250,212
8005 DATA 6,63,62,255,174
8006 DATA 119,35,16,249,24
8007 DATA 39,33,250,212,6
8008 DATA 63,197,126,1,0
8009 DATA 8,167,203,31,203

8010 DATA 17,16,250,113,193
8011 DATA 35,16,239,6,21
8012 DATA 33,250,212,17,36
8013 DATA 213,78,26,119,121
8014 DATA 18,35,19,16,247
8015 DATA 33,250,212,205,87
8016 DATA 212,201,33,14,213
8017 DATA 17,250,212,6,3
8018 DATA 197,213,229,6,10
8019 DATA 78,26,119,121,18

8020 DATA 43,19,16,247,225
8021 DATA 209,1,21,0,84
8022 DATA 93,19,9,193,16
8023 DATA 230,24,213,0,0
8024 DATA 0,0,0,0,255
8025 DATA 255,255,0,0,190
8026 DATA 190,190,190,190,190
8027 DATA 190,190,190,190,128
8028 DATA 254,254,254,254,254
8029 DATA 254,254,254,0,0
8030 DATA 190,0,0,0,0
```

DISPLAYING SPRITES

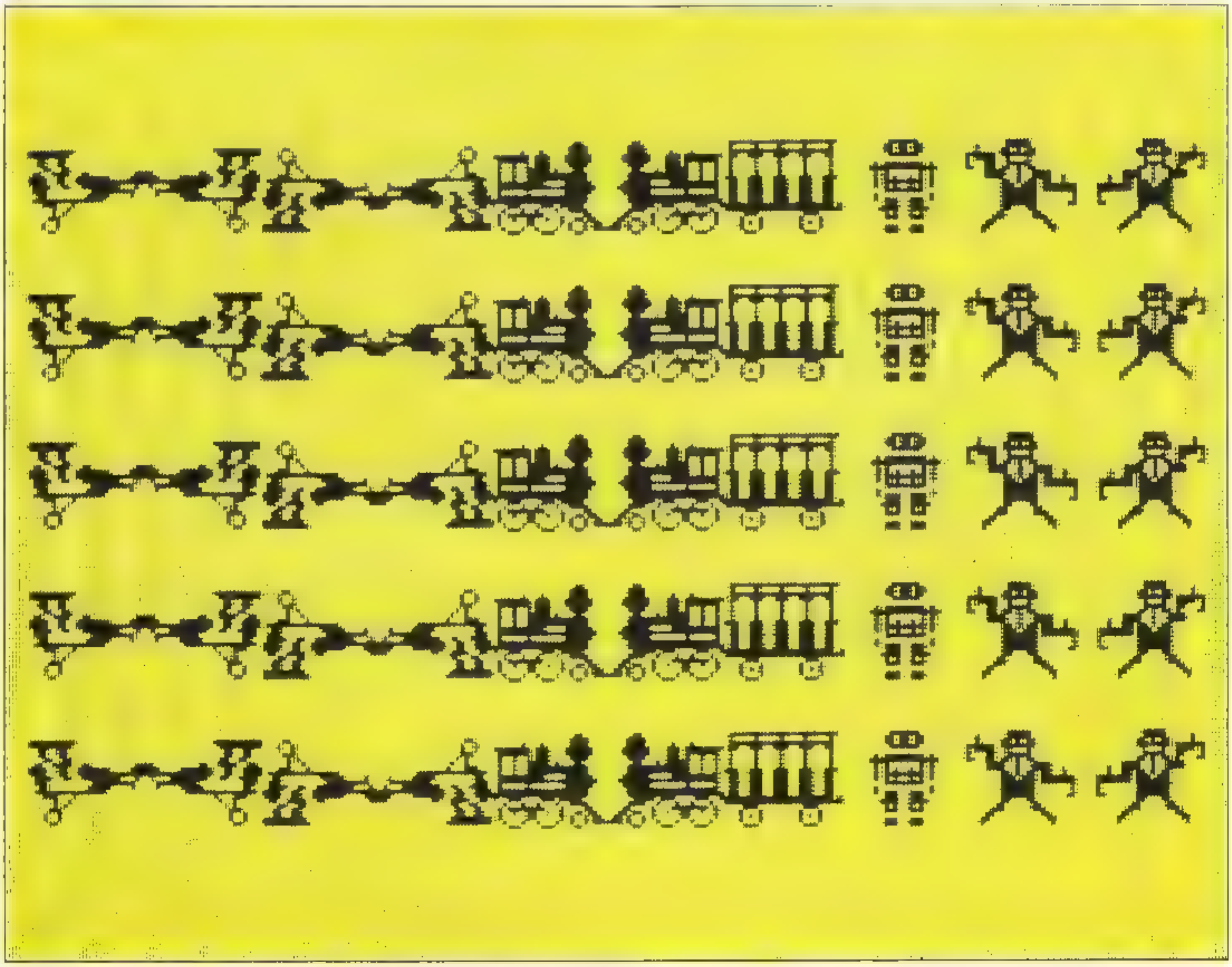
Once your sprites are defined it is very useful to be able to see what they look like on the screen. Obviously this can be done using the sprite-handling routine, but it is very hard to examine a sprite with a critical eye while it is moving across the screen! To avoid this problem, and also to give you the chance of looking at the shapes and designs of several sprites at once, the sprite print routine, FNf, has been provided.

This routine prints a sprite from the sprite buffer onto the screen at a specified position. Since any of the sprites can be printed at any position on the screen you can use the routine to preview the sprites you have designed, and it is at this stage that you can decide the best sort of starting and finishing positions for the sprites when they come to be used.

You can also use the routine to preview the effects of animation by calling the routine repeatedly to print sprites in an animation sequence on top of one another.

SPRITE DISPLAY PROGRAM

```
  10 DEF FN f(x,y,n)=USR 54100
 100 BORDER 4: PAPER 4: INK 0: C
LS
 110 FOR n=1 TO 10
 120 FOR k=1 TO 5
 130 RANDOMIZE FN f(n*3-3,(k*4)-
3,n)
 140 NEXT k
 150 NEXT n
 160 PAUSE 0
 500 CLS
 510 RANDOMIZE FN f(10,10,9)
 520 PAUSE 10
 530 RANDOMIZE FN f(10,10,10)
 540 PAUSE 10
 550 GO TO 510


0 OK, 0:1
```



This will give you an idea of how effective the animation is going to be, as well as allowing you to make any changes to the sprites before you work out the animation in detail.

How sprites are stored

Before examining the machine-code routine it is important to understand how sprites are stored in memory. As you know, a sprite table which stores ten sprites can be found at location 54600 onwards in memory. A 24 by 21 sprite has 504 elements (24x21), and since any of these elements can have an ink attribute either set or not set, this means that 504 bits of information need to be stored away for each sprite. Each sprite requires 63 bytes of memory (since there are eight bits in a byte) and, as all ten sprites are stored one after the other, they take up a total of 630 bytes.

Each sprite is arranged in memory as shown in the diagram below. Numbers correspond to the byte numbers (0-63), and the 1s and 0s represent the individual bits.

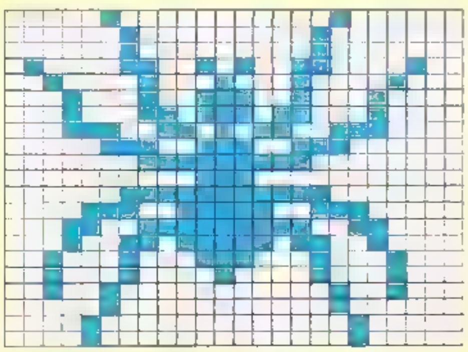
HOW SPRITES ARE STORED IN BINARY

Sprites are stored as binary numbers in 63 bytes (504 bits) of memory. The 1s and 0s correspond to set and unset pixels respectively on the sprite diagram shown underneath.

how memory bits are set

```
1    0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
25   0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
49   0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
73   0 1 0 0 0 0 1 1 0 0 1 0 1 0 0 1 1 0 0 0 0 1 0 0
97   0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0
121  0 0 0 1 0 0 0 1 0 0 1 1 1 0 0 1 0 0 0 1 0 0 0 0
145  0 0 0 1 0 0 0 1 1 0 1 1 1 0 1 1 0 0 0 1 0 0 0 0
160  0 0 0 1 1 1 0 0 1 1 0 1 0 1 1 0 0 1 1 1 0 0 0 0
193  0 0 0 0 0 1 1 1 0 0 1 1 1 0 0 1 1 1 0 0 0 0 0 0
217  0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0
241  0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0
265  0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0
289  0 0 0 0 1 1 1 0 0 1 1 1 1 1 0 0 1 1 1 0 0 0 0 0
313  0 0 0 1 1 0 0 1 1 1 1 1 1 1 1 1 0 0 1 1 0 0 0 0
337  0 0 0 1 0 0 1 1 0 1 1 1 1 1 0 1 1 0 0 1 0 0 0 0
361  0 0 1 0 0 0 1 0 0 0 1 1 1 0 0 0 1 0 0 0 1 0 0 0
385  0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0
409  0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0
433  0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
457  0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
481  0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
```

sprite diagram



The BASIC program

The program on this page shows the contents of a sprite buffer, containing ten different sprites, displayed on the screen. The program calls the sprite print routine five times in a loop, to display the sprites down the screen as well as across it.

FNf

SPRITE PRINT ROUTINE

Start address 54100 **Length** 75 bytes
Other routines called Sprite editor routines (FNa-FNe).
What it does Prints a single sprite on the screen at a specifed point.

Using the routine This routine displays any single sprite from the sprite buffer. The routine does not move the sprite. Note that if the sprite is too far to the right of the screen it will reappear one character below on the left-hand side of the screen, since the Spectrum PRINT routine is used in the transfer of memory to screen.

ROUTINE PARAMETERS

DEF FNf(x,y,n)	
x,y	specify print position (x<29, y<21)
n	specifies number of sprite (1-10)

ROUTINE LISTING

```
7900 LET b=54100: LET l=70: LET
z=0: RESTORE 7910
7901 FOR i=0 TO l-1: READ a
7902 POKE (b+i),a: LET z=z+a
7903 NEXT i
7904 LET z=INT (((z/l)-INT (z/l)
)*l)
7905 READ a: IF a<>z THEN PRINT
"??": STOP

7910 DATA 42,11,92,1,4
7911 DATA 0,9,86,1,8
7912 DATA 0,9,94,237,83
7913 DATA 143,211,9,126,50
7914 DATA 150,211,123,230,24
7915 DATA 246,64,103,123,230
7916 DATA 7,183,31,31,31
7917 DATA 31,130,111,34,146
7918 DATA 211,58,150,211,71
7919 DATA 17,63,0,33,9
7920 DATA 213,25,16,253,237
7921 DATA 91,146,211,195,1
7922 DATA 212,201,174,72,13
7923 DATA 14,3,0,0,0
7924 DATA 60,0,0,0,0
```

How the program works

Line 10 defines the sprite print routine, which has three parameters: the x,y screen co-ordinates at which the sprite is to be printed, and the number of the sprite (from the ten sprites stored in the sprite table). Lines 90 and 180 set up loops to print the sprites. Line 190 prints the sprite, using a combination of the two loop variables to define the x,y co-ordinate.

Lines 300 to 330 show how animation can be achieved using a print routine. A left and right mirror image of a single sprite are printed one after the other to produce a simple animation effect. You could produce more complex animation in this way; the only limitation to the method is the amount of time it takes to define sprites, although the animation in this program can also be speeded up substantially by leaving out the PAUSE statements. Of course, the other drawback is that you cannot move the sprite around the screen using this method.

MOVING SPRITES 1

Now that you have routines to create sprites and display them on the screen, you need a routine which makes the sprites change position. The routines introduced here enable you to get your sprites moving on the screen.

The master sprite routine

This routine enables you to move simple (that is, not animated) sprites. The routine has no title since it is always used together with the sprite-controlling routines in this book; by itself, the routine does nothing. When you use a sprite, it is this routine which causes the sprite to move across the screen in the required way. The main job of the routine is to print a sprite on the screen using Exclusive/OR printing, wipe it off, and print it again one pixel away until the program or routine asks the sprite to stop moving.

The routine has been programmed to work whether it has been called by a routine which is working within BASIC (a normal routine) or by one working independently of BASIC (an interrupt-driven routine).

The sprite-handling routine

This routine, FNg, allows you to control the movement of sprites. The routine works in conjunction with the master sprite routine, and both must be present in memory for sprites to move on the screen. The routine has a range of parameters to control exactly how the sprite will appear on the screen.

How to use the routines

Having produced some sprites, it is a simple matter to display them against a background, and the ideal way of creating backgrounds is to use a graphics editor program such as that in Book Three. All the background displays in this book were created using the graphics editor. The programs in this book do not themselves create backgrounds; you must add these yourself.

Interrupts

Most machine-code routines are called (as you will have seen) from BASIC, and are then executed in much the

MASTER SPRITE ROUTINE

Start address 53700 **Length** 365 bytes

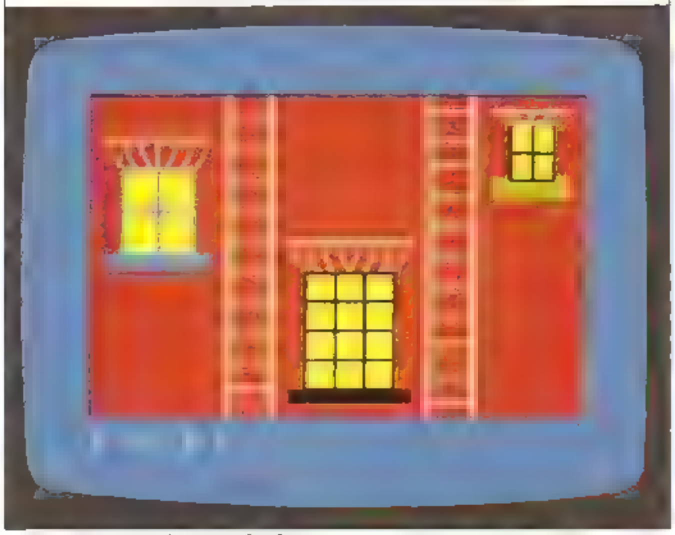
What it does Used in conjunction with the sprite-handling routine (FNg), this routine takes a sprite from the sprite table at location 54600 and moves it across the screen.

Using the routine This routine must always be used together with the other sprite routines given in this book; if used by itself, you will not see anything happen on the screen. Whenever sprites are used, the master sprite routine is called by the other routines to do the work of moving the sprite.

ROUTINE LISTING

```
7800 LET b=53700: LET l=360. LET
 z=0: RESTORE 7810
7801 FOR i=0 TO l-1: READ a
7802 POKE (b+i),a: LET z=z+a
7803 NEXT i
7804 LET z=INT (((z/l)-INT (z/l)
)*l)
7805 READ a: IF a<>z THEN PRINT
"??": STOP

7810 DATA 229,213,197,245,229
7811 DATA 221,225,120,205,177
7812 DATA 34,40,82,237,67
7813 DATA 15,210,50,229,209
7814 DATA 62,21,8,221,94
7815 DATA 0,221,86,21,221
7816 DATA 78,42,6,1,175
7817 DATA 203,27,203,26,203
7818 DATA 25,31,16,247,35
7819 DATA 35,35,174,119,43

7820 DATA 126,169,119,43,126
7821 DATA 170,119,43,126,171
7822 DATA 119,8,61,40,25
7823 DATA 8,221,35,36,124
7824 DATA 230,7,32,205,1
7825 DATA 49,0,8,95,8
7826 DATA 62,21,147,128,71
7827 DATA 205,177,34,24,189
7828 DATA 241,193,209,225,201
7829 DATA 237,67,81,210,62

7830 DATA 21,8,221,94,0
7831 DATA 221,86,21,221,78
7832 DATA 42,35,35,126,169
7833 DATA 119,43,126,170,119
7834 DATA 43,126,171,119,8
7835 DATA 61,40,218,8,221
7836 DATA 35,36,124,230,7
7837 DATA 32,221,8,95,8
7838 DATA 1,208,46,62,21
7839 DATA 147,128,71,205,177

7840 DATA 34,24,205,229,213
7841 DATA 197,229,221,225,151
7842 DATA 50,41,211,120,205
7843 DATA 177,34,202,216,210
7844 DATA 237.67,194,210,50
7845 DATA 130,210,62,21,8
7846 DATA 221,94,0,221,86
7847 DATA 21,221,78,42,6
7848 DATA 1,175,203,27,203
7849 DATA 26,203,25,31,16

7850 DATA 247,35,35,35,71
7851 DATA 126,160,196,33,211
7852 DATA 126,168,119,43,126
7853 DATA 161,196,33,211,126
7854 DATA 169,119,43,126,162
7855 DATA 196,33,211,126,170
7856 DATA 119,43,126,163,196
7857 DATA 33,211,128,171,119
7858 DATA 8,61,40,25,8
7859 DATA 221,35,36,124,230

7860 DATA 7,32,183,1,49
7861 DATA 0,8,95,8,62
7862 DATA 21,147,128,71,205
7863 DATA 177,34,24,167,193
7864 DATA 209,225,58,41,211
7865 DATA 201,237,67,21,211
7866 DATA 62,21,8,221,94
7867 DATA 0,221,86,21,221
7868 DATA 78,42,35,35,126
7869 DATA 161,196,33,211,126

7870 DATA 169,119,43,126,162
7871 DATA 195,33,211,126,170
7872 DATA 119,43,126,163,196
7873 DATA 33,211,126,171,119
7874 DATA 8,61,40,201,8
7875 DATA 221,35,36,124,230
7876 DATA 7,32,206,8,95
7877 DATA 8,1,208,46,62
7878 DATA 21,147,128,71,205
7879 DATA 177,34,24,190,245
7880 DATA 62,1,50,41,211
7881 DATA 241,201,1,0,0
7882 DATA 83,0,0,0,0
```

TYPICAL SPRITE BACKGROUND



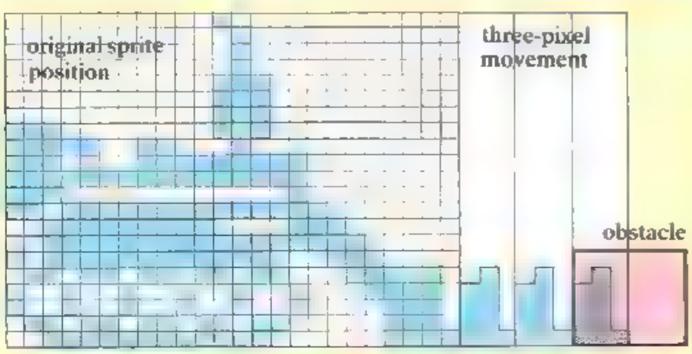
same way as the BASIC program, line by line. When the machine-code routine has finished running control is returned to the BASIC program. However, there is a much more sophisticated method of using machine code, which is to make a routine run independently of BASIC.

This can be achieved by taking advantage of the fact that at regular intervals the Spectrum interrupts the running of any BASIC program or machine-code routine which is being executed. It does this so that it can print information on the screen, and perform various other household chores, like memory management. These interrupts occur at least fifty times a second, so quickly in fact that if you carry out a machine-code routine as well as a BASIC program using interrupts it will appear that both things are happening simultaneously.

Two of these "interrupt-driven" routines have been provided in this book. One is the keyboard-controlled sprite, and the other is the interrupt-driven window given on pages 32-33.

HOW A SPRITE MOVES

Sprites move in blocks of three pixels. On reaching an obstacle, and with the collision flag set to 1, a sprite continues to move for three pixels before stopping.



FNg

SPRITE-HANDLING ROUTINE

Start address 53500 **Length** 170 bytes
Other routines called Master sprite routine.
What it does Prints and moves a single sprite on the screen.

Using the routine The screen area is measured in pixel co-ordinates from the top left-hand corner, rather than from the bottom of the screen.
Sprites move in multiples of three pixels, so a value of 60 for l moves the sprite 180 pixels. If the collision detection flag is set to 0, the sprite will pass over obstacles (any pixels with INK set). If c is set to 1, the sprite will stop when it hits an object (after an overlap of three pixels). You can find out the precise position where a sprite stopped by PEEKing locations 53498 and 53499 for the y and x co-ordinates respectively.

ROUTINE PARAMETERS

DEF FNg(x,y,d,l,s,c,n)	
x,y	specify top left-hand corner of sprite (x<232, y<155)
d	direction of travel of the sprite (0=left, 1=right, 2=up, 3=down)
l	distance moved (vertical<=51, horizontal<=77)
s	switch (s=1 to disappear, s=0 to remain on screen)
c	collision detection flag (1=stop, 0=continue)
n	specifies number of sprite (1-10)

ROUTINE LISTING

```
7750 LET b=53500: LET l=165: LET
 z=0: RESTORE 7760
7751 FOR i=0 TO l-1: READ a
7752 POKE (b+i),a: LET z=z+a
7753 NEXT i
7754 LET z=INT (((z/l)-INT (z/l)
)*l)
7755 READ a: IF a<>z THEN PRINT
"???": STOP

7760 DATA 42,11,92,17,4
7761 DATA 0,25,75,30,8
7762 DATA 25,70,25,126,230
7763 DATA 3,50,158,209,25
7764 DATA 126,50,159,209,25
7765 DATA 126,230,1,50,160
7766 DATA 209,25,126,230,1
7767 DATA 50,161,209,25,126
7768 DATA 17,63,0,33,9
7769 DATA 213,25,51,32,252

7770 DATA 205,93,210,254,0
7771 DATA 40,19,58,161,209
7772 DATA 254,0,40,12,58
7773 DATA 160,209,198,1,230
7774 DATA 1,50,160,209,24
7775 DATA 71,118,205,196,209
7776 DATA 58,158,209,254,0
7777 DATA 40,19,254,1,40
7778 DATA 26,254,2,40,33
7779 DATA 5,5,5,120,230

7780 DATA 252,40,44,195,132
7781 DATA 209,13,13,13,121
7782 DATA 230,252,40,33,195
7783 DATA 132,209,12,12,12
7784 DATA 121,214,231,48,22
7785 DATA 195,132,209,4,4
7786 DATA 4,120,214,150,48
7787 DATA 11,58,159,209,61
7788 DATA 50,159,209,254,0
7789 DATA 32,159,58,160,209
7790 DATA 237,67,250,208,254
7791 DATA 0,200,118,205,93
7792 DATA 210,201,1,47,0
7793 DATA 66,0,0,0,0
```

MOVING SPRITES 2

The sprite-handling routine has many user-controlled features built into it, as you can see from the long list of parameters which are passed to it. It is a good idea to become familiar with these, as otherwise you will under-utilize the potential of this very powerful routine. The train program, given here, is a good example of how the parameters are used.

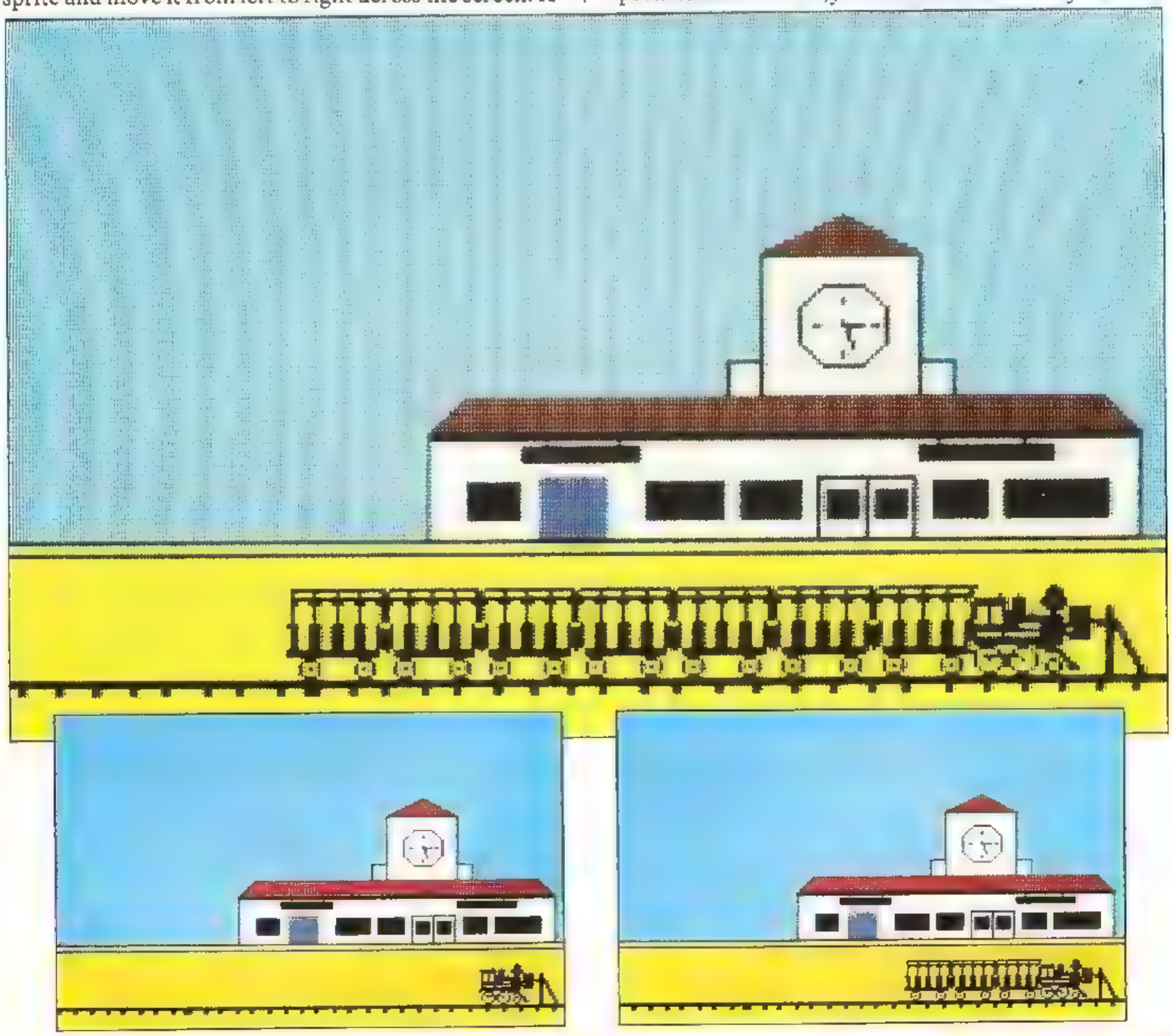
The train program

The large display on this page shows a train and some carriages being shunted along a railway line. All the movement in this program is controlled by the sprite-handling routine. Lines 110 and 120 take the train sprite and move it from left to right across the screen. A mirror image of the train is then selected and driven back to the start position, where it remains on the screen. Lines 130-150 set up a loop which calls carriages one by one. Since this contains the collision flag set to 1 each carriage moves across the screen until it encounters an obstacle—the previous carriage — when it stops and remain on the screen.

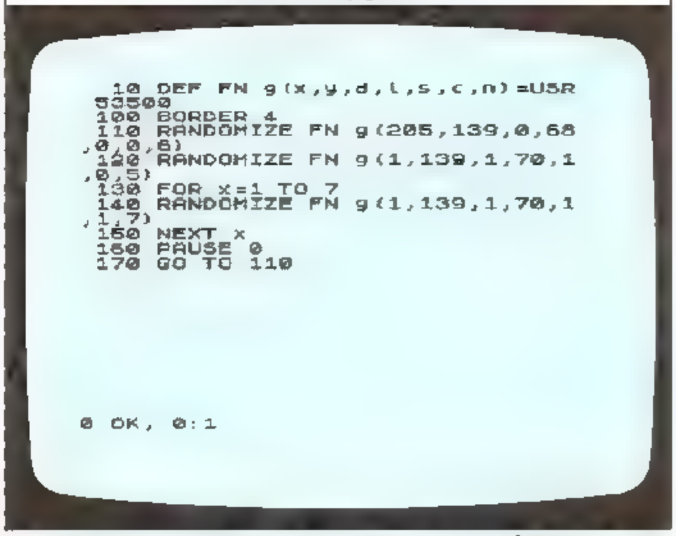
More about the parameters

It is worth looking at the impressive range of parameters available with the sprite-handling routine in a little more detail.

The first feature which you will use, of course, is the specification of the x,y co-ordinate at which you want



TRAIN PROGRAM



the sprite to start. Normally you can specify this simply by looking at the screen, but more sophisticated methods are available for determining the start point.

In the program example a train appears from the left and crosses to the right-hand side. It travels a distance of 65 (that is, 195 pixels) and so you know that you can start the second train off from a point 195 pixels to the right of where the first started. But what if you didn't know where the first train had started or stopped? Since the routine stores the last pixel position of the sprite at two pointers in memory, the new sprite could then start from position (PEEK(53499),(PEEK(53498)).

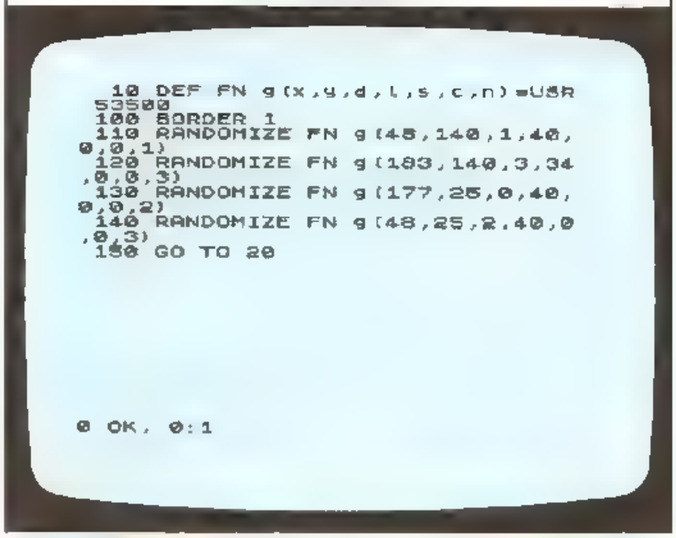
There is a case for making the distance moved by the sprite (parameter l) as short as possible with each call to the routine, despite this producing some flicker because of the volume of transfers from BASIC to machine code and back. This is because while you are in BASIC you can keep a check on the current screen, the position of the sprite and so on, but while the sprite is moving you do not have any control over it. This, of course, is an advantage of interrupt-driven routines which allow you to monitor the progress of other things on the screen while a sprite or window is moving.

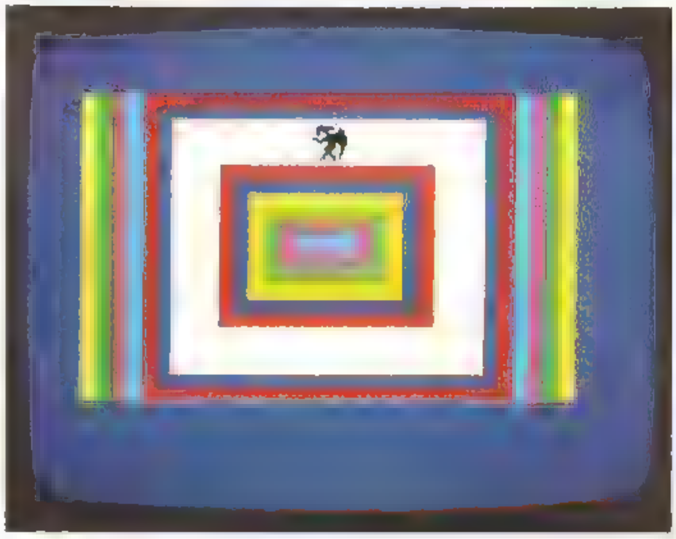
The value you give the switch, s, depends on what you want the sprite to do after you have finished using it. In the program the switch is off for the first train and on for the second: after the first train has come on and travelled across the screen it should disappear before the second returns. On the other hand the second train must remain on the screen after its journey as, if it doesn't, the carriage will have nothing to run into.

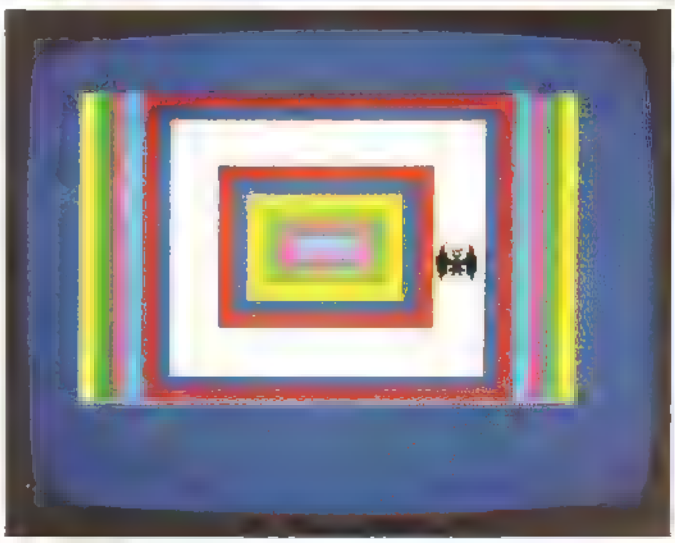
The bat program

The final program on this page shows how the sprite-handling routine can be called a number of times in a loop to move an object in four different directions around the screen. Although you see the bat moving continuously on the screen, the program listing reveals that four different routine calls are being used, one for each direction that the sprite is being moved, and three different sprite shapes are used to give various views of the bat's body as it moves around the loop.

BAT PROGRAM







KEYBOARD-CONTROLLED SPRITES

The keyboard-controlled sprite routine, FNh, enables you to control the movement of sprites using the cursor keys. The only difficulty with the routine lies not so much in using it as in switching it off. Because the routine is interrupt-driven it continues to operate after any BASIC program has finished. Even as you edit a program you will find that the sprite is still moving on the screen. A subroutine is required to switch it off:

```
2000 DEF FN z(s)=USR 53100
2010 RANDOMIZE FN z(0)
2020 RETURN
```

This redefines the routine with just one parameter, the switch. If the switch is given a value of 0, the routine will be switched off. The maze program on page 20 gives you a chance to try using the routine; you can create your own maze with the Book Three graphics editor.

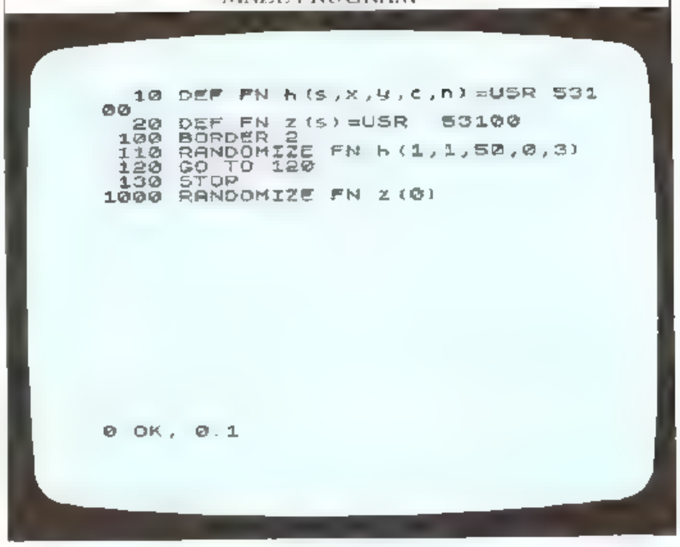
Controlling the routine

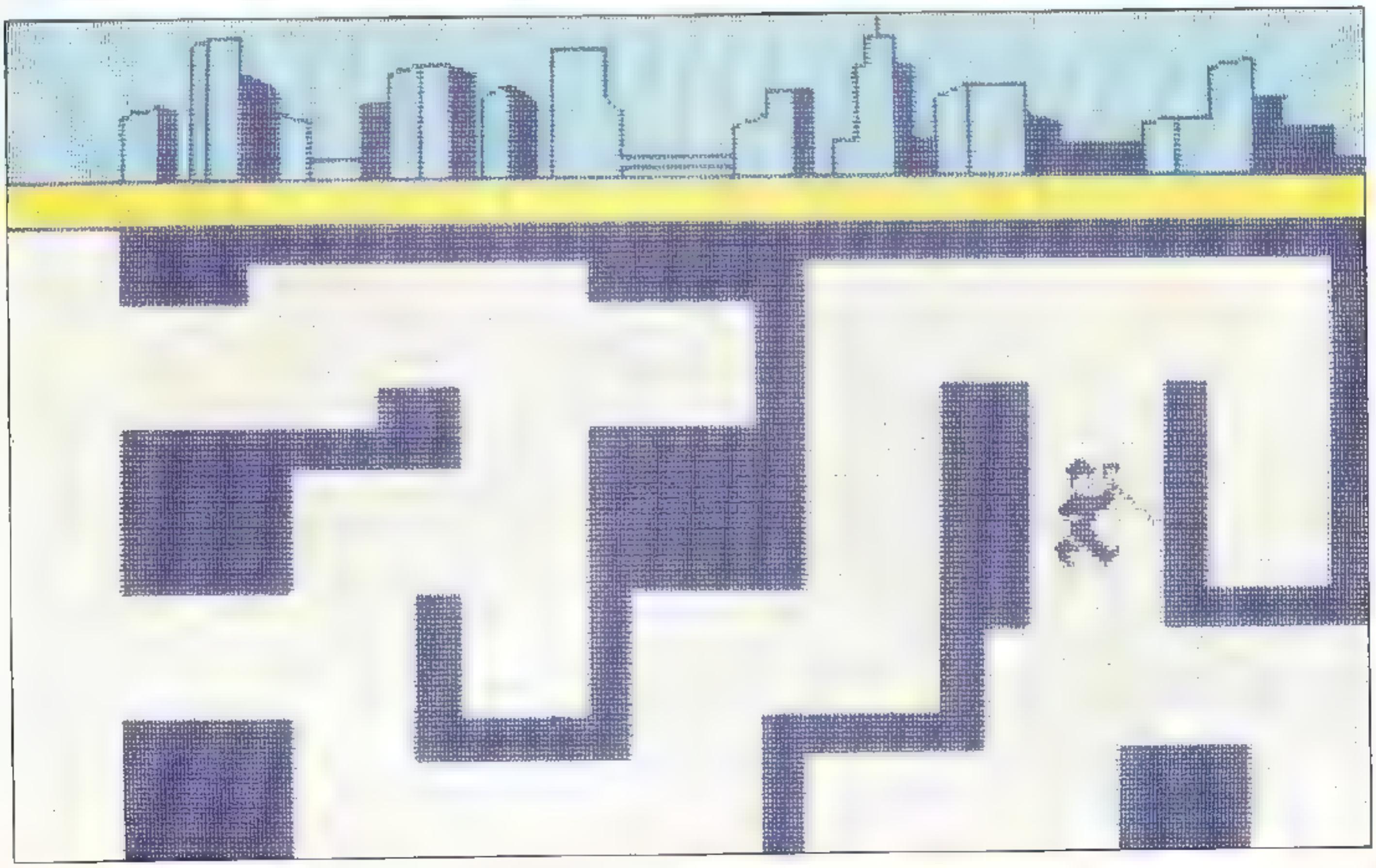
Although keyboard-controlled, the routine is quite hard to control from within a program. Since the sprite routine runs outside BASIC, it is only when the routine stops that you can check its position. One solution is to leave the collision detection off but to set up your own collision detection instead. You can do this by switching the sprite off, and looking at the last x,y co-ordinate (stored at 53099, 53098). Then use the POINT command to find if pixels at these co-ordinates are set:

x,y	x+24,y
x,y+21	x+24,y+21

If this is so, a collision has occurred.

MAZE PROGRAM





FNh

KEYBOARD-CONTROLLED SPRITE ROUTINE

Start address 53100 **Length** 250 bytes
Other routines called Master sprite routine.
What it does Puts a sprite on the screen and allows it to be controlled by the cursor keys.

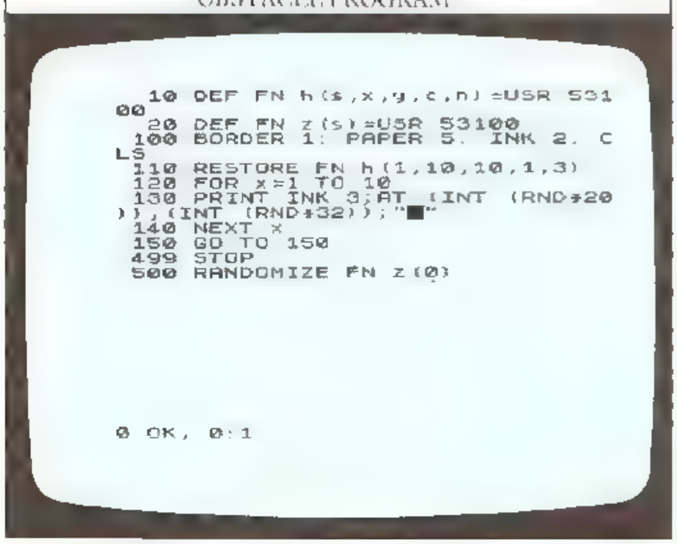
Using the routine The routine is interrupt-driven, so it will continue to respond to keyboard presses until switched off by calling the routine with the switch parameter (s) set to 0.
If not switched off, the routine continues to operate even when you stop the program and attempt to, say, edit — you will see a sprite moving whenever you use the cursor keys. It is advisable to switch the routine off at the end of any BASIC program which calls the keyboard-controlled sprite routine. This can be done by defining the function a second time using a function title not used elsewhere in the program, say FNz. This function is defined as having a single parameter only, s, which means that the function can then be called with this parameter only, set to zero. The collision detection parameter, c, can be used to detect if the sprite passes over any pixels with set INK attributes.

ROUTINE PARAMETERS	
DEF FN h(s,x,y,c,n)	
s	switch (1=on, 0=off)
x,y	start position of sprite (0<=x<=231, 0<=y<154)
c	collision detection (1=on, 0=off)
n	number of sprite (1-10)

The obstacle program

The final program again has the aim of avoiding obstacles. Lines 10 and 20 define the routine twice as before. Line 110 switches the keyboard sprite on. Lines 120-140 print a series of random graphics blocks on the screen. The aim of the game is to avoid these blocks. Line 500 is a continuous loop which keeps the program RUNning.

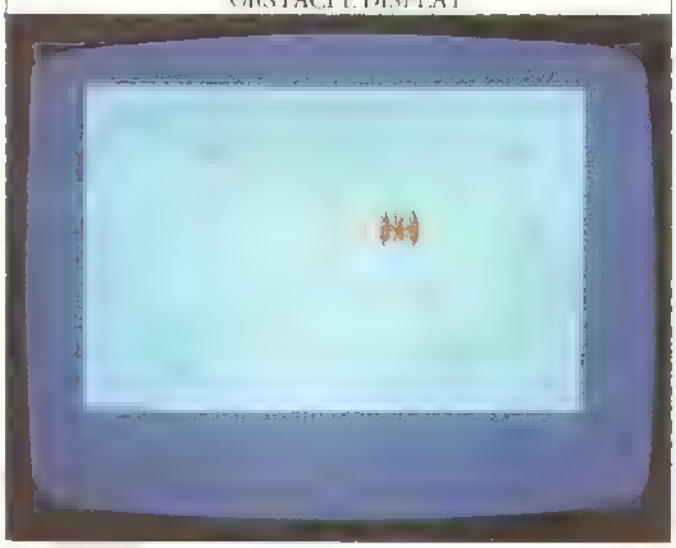
OBSTACLE PROGRAM



ROUTINE LISTING

```
7650 LET b=53100: LET l=245: LET
 z=0: RESTORE 7660
7651 FOR i=0 TO l-1: READ a
7652 POKE (b+i),a: LET z=z+a
7653 NEXT i
7654 LET z=INT (((z/l)-INT (z/l)
)*l)
7655 READ a: IF a<>z THEN PRINT
"??": STOP

7660 DATA 42,11,92,17,4
7661 DATA 0,25,126,230,1
7662 DATA 50,95,208,32,11
7663 DATA 42,96,208,237,75
7664 DATA 106,207,205,196,209
7665 DATA 201,30,8,25,78
7666 DATA 25,70,25,126,230
7667 DATA 1,50,93,208,25
7668 DATA 126,33,9,213,17
7669 DATA 63,0,25,61,32

7670 DATA 252,34,96,208,205
7671 DATA 93,210,237,67,106
7672 DATA 207,243,175,71,50
7673 DATA 94,208,62,207,33
7674 DATA 2,206,119,35,16
7675 DATA 252,62,206,237,71
7676 DATA 237,94,251,201,0
7677 DATA 0,0,0,0,0
7678 DATA 0,0,0,0,0
7679 DATA 0,0,0,0,255

7680 DATA 243,245,197,213,229
7681 DATA 8,245,8,221,229
7682 DATA 58,95,208,254,0
7683 DATA 40,24,205,7,208
7684 DATA 58,94,208,254,0
7685 DATA 40,14,58,93,208
7686 DATA 254,0,40,7,205
7687 DATA 123,207,175,50,95
7688 DATA 208,195,252,207,2[illegible]1
7689 DATA 225,8,241,8,225

7690 DATA 209,193,241,251,201
7691 DATA 237,75,106,207,42
7692 DATA 96,208,62,239,219
7693 DATA 254,203,103,40,44
7694 DATA 203,95,40,31,203
7695 DATA 87,40,16,62,247
7696 DATA 219,254,203,103,4[illegible]
7697 DATA 1,201,13,13,13
7698 DATA 121,230,252,200,24
7699 DATA 25,12,12,12,62

7700 DATA 231,145,216,24,16
7701 DATA 5,5,5,120,230
7702 DATA 252,200,24,7,4
7703 DATA 4,4,62,153,144
7704 DATA 216,197,237,75,106
7705 DATA 207,205,196,209,193
7706 DATA 205,93,210,50,94
7707 DATA 208,237,67,106,207
7708 DATA 201,1,0,0,9
7709 DATA 26,0,0,0,0
```

OBSTACLE DISPLAY



DOUBLE-SIZED SPRITES

You have already seen what can be done with a sprite 24 by 21 pixels (504 pixels in all), but there are occasions when you would like to use larger sprites still. This makes great demands upon your Spectrum, but the two routines that are provided here (FNi and FNj) each give you the power to move over 1000 pixels at once. These routines provide you with double horizontal and vertical sprites respectively. In each case sprites from the sprite table are attached to one another and are then moved together in exactly the same way as a single sprite — though naturally not quite as quickly.

The double sprite programs

Both the demonstration programs are straightforward. In the first program, a double horizontal sprite has been used, and this demonstrates the effectiveness of quite large moving objects — it would take 18 user-defined graphics to define the area of the car, let alone move it! The program enables you to see the great improvement in sprite visibility that can be obtained by doubling its size.

FNi

DOUBLE HORIZONTAL SPRITE ROUTINE

Start address 52400 **Length** 235 bytes
Other routines called Sprite editor routines (FNa-FNe).
What it does Displays and moves two sprites together horizontally on the screen.

Using the routine The routine takes sprites n (left) and n+1 (right) from the sprite buffer. Parameters are as for the sprite-handling routine (FNg). Bytes 52398 and 52399 specify the y and x co-ordinates of the sprite's final position after calling the routine.

ROUTINE PARAMETERS

DEF FNi(x,y,d,l,s,c,n)

x,y	start co-ordinates (0<=x<=231,0<=y<=154)
d	specifies direction of travel (0—left, 1—right, 2—up, 3—down)
l	distance moved (vert<=51, horiz<=77)
s	switch (1—on, 0—off)
c	flag for collision detection (1—on, 0—off)
n	number of first sprite (1-10)

ROUTINE LISTING

```
7550 LET b=52400: LET l=230: LET
 z=0. RESTORE 7560
7551 FOR i=0 TO l-1: READ a
7552 POKE (b+i),a: LET z=z+a
7553 NEXT i
7554 LET z=INT (((z/l)-INT (z/l)
)*l)
7555 READ a: IF a<>z THEN PRINT
"??": STOP
```

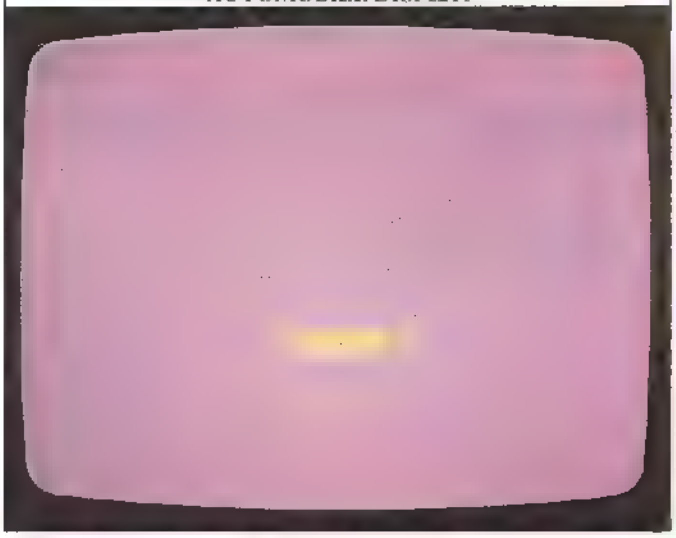
The two sprites which make up the car are stored as the first and second of the sprite table — the routine simply takes the sprite specified by the n parameter, together with the following sprite from the sprite table that is stored in memory.

AUTOMOBILE PROGRAM

```
  10 DEF FN i(x,y,d,l,s,c,n)=USR
 52400
 100 BORDER 3: PAPER 3: INK 6: C
LS
 110 RANDOMIZE FN i(10,130,1,65,
0,0,1)
 120 RANDOMIZE FN i(205,130,0,65
,0,0,3)
 130 GO TO 110

0 OK, 0:1
```

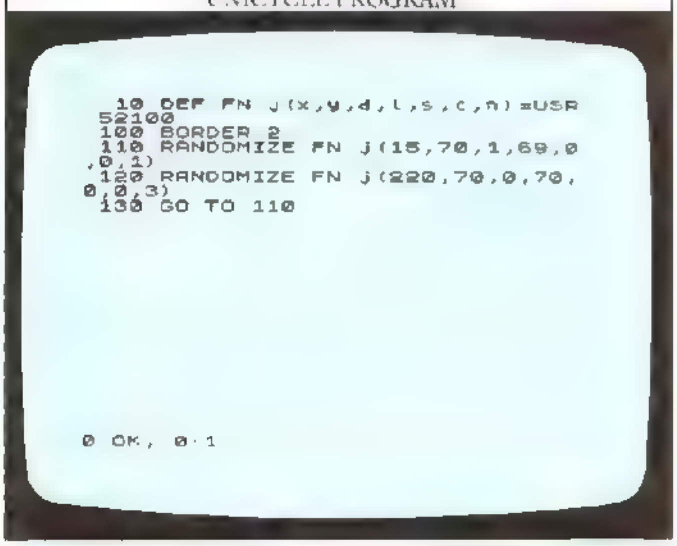
```
7560 DATA 42,11,92,17,4
7561 DATA 0,25,78,30,8
7562 DATA 25,70,25,126,230
7563 DATA 3,50,148,205,25
7564 DATA 126,50,149,205,25
7565 DATA 126,230,1,50,150
7566 DATA 205,25,126,230,1
7567 DATA 50,146,205,25,126
7568 DATA 17,63,0,33,9
7569 DATA 213,25,61,32,252

7570 DATA 175,50,147,205,205
7571 DATA 93,210,205,137,205
7572 DATA 197,229,17,63,0
7573 DATA 25,62,24,129,79
7574 DATA 205,93,210,205,137
7575 DATA 205,225,193,197,229
7576 DATA 6,1,118,16,253
7577 DATA 225,193,0,0,118
7578 DATA 205,196,209,197,229
7579 DATA 17,63,0,25,62

7580 DATA 24,129,79,205,196
7581 DATA 209,225,193,58,148
7582 DATA 205,254,0,40,19
7583 DATA 254,1,40,26,254
7584 DATA 2,40,33,5,5
7585 DATA 5,120,230,252,40
7586 DATA 59,195,84,205,13
7587 DATA 13,13,121,230,252
7588 DATA 40,48,195,84,205
7589 DATA 12,12,12,121,214

7590 DATA 207,48,37,195,84
7591 DATA 205,4,4,4,120
7592 DATA 214,150,48,26,58
7593 DATA 147,205,254,0,40
7594 DATA 7,58,146,205,254
7595 DATA 0,32,12,58,149
7596 DATA 205,61,50,149,205
7597 DATA 254,0,194,230,204
7598 DATA 58,150,205,237,67
7599 DATA 174,204,254,0,200

7600 DATA 118,205,93,210,17
7601 DATA 63,0,25,62,24
7602 DATA 129,79,118,205,93
7603 DATA 210,201,254,0,200
7604 DATA 62,1,50,147,205
7605 DATA 201,1,0,1,0
7606 DATA 53,0,0,0,0
```

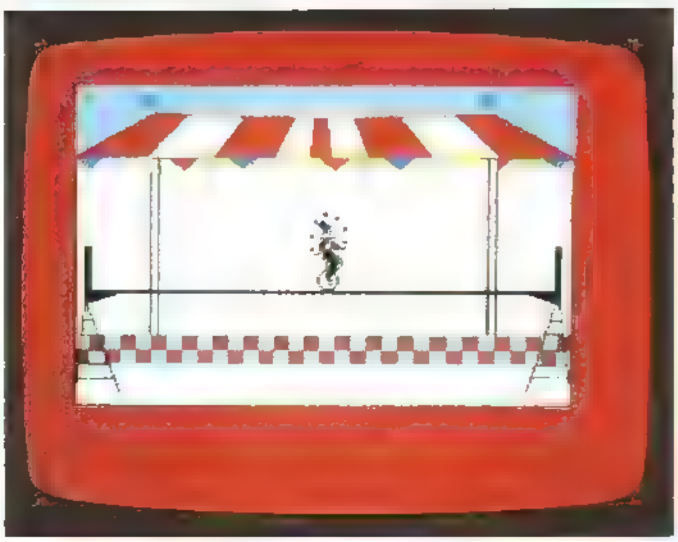
AUTOMOBILE DISPLAY



The other program shows a double vertical sprite — a unicyclist — against a circus background, drawn using the graphics editor from Book Three.

UNICYCLE PROGRAM





FNj

DOUBLE VERTICAL SPRITE ROUTINE

Start address 52100 **Length** 230 bytes
What it does Displays a double vertical sprite.

Using the routine Used in the same way as the horizontal sprite routine, but final sprite position now specified by 52098 and 52099.

ROUTINE PARAMETERS

DEF FNj(x,y,d,l,s,c,n)	
x,y	start co-ordinates (0<=x<=231,0<=y<=154)
d	specifies direction of travel (0=left, 1=right, 2=up, 3=down)
l	distance moved (vertical<=51, horizontal<=77)
s	switch (1=on, 0=off)
c	flag for collision detection (1=on, 0=off)
n	number of first sprite(1-10)

ROUTINE LISTING

```
7450 LET b=52100: LET l=225: LET
 z=0: RESTORE 7460
7451 FOR i=0 TO l-1: READ a
7452 POKE (b+i),a: LET z=z+a
7453 NEXT i
7454 LET z=INT (((z/l)-INT (z/l)
)*l)
7455 READ a: IF a<>z THEN PRINT
"??": STOP

7460 DATA 0,42,11,92,17
7461 DATA 4,0,25,78,30
7462 DATA 8,25,70,25,12[illegible]
7463 DATA 230,3,50,103,204
7464 DATA 25,126,50,104,204
7465 DATA 25,126,230,1,50
7466 DATA 105,204,25,12[illegible],230
7467 DATA 1,50,101,204,25
7468 DATA 126,17,63,0,33
7469 DATA 9,213,25,61,3[illegible]

7470 DATA 252,175,50,102,204
7471 DATA 205,93,210,205,86
7472 DATA 204,197,229,17,63
7473 DATA 0,25,62,21,128
7474 DATA 71,205,93,210,205
7475 DATA 88,204,225,193,197
7476 DATA 229,6,1,118,16
7477 DATA 253,225,193,0,0
7478 DATA 118,205,196,209,197
7479 DATA 229,17,63,0,25

7480 DATA 62,21,128,71,205
7481 DATA 196,209,225,193,58
7482 DATA 103,204,254,0,40
7483 DATA 19,254,1,40,26
7484 DATA 254,2,40,33,5
7485 DATA 5,5,120,230,252
7486 DATA 40,51,195,41,204
7487 DATA 13,13,13,121,230
7488 DATA 252,40,40,195,41
7489 DATA 204,12,12,12,121

7490 DATA 214,231,48,29,195
7491 DATA 41,204,4,4,4
7492 DATA 120,214,132,48,18
7493 DATA 58,102,204,254,0
7494 DATA 32,11,58,104,204
7495 DATA 61,50,104,204,254
7496 DATA 0,32,128,58,105
7497 DATA 204,237,67,202,91
7498 DATA 254,0,200,118,205
7499 DATA 93,210,17,63,0

7500 DATA 25,62,21,128,71
7501 DATA 118,205,93,210,201
7502 DATA 254,0,200,58,101
7503 DATA 204,254,0,200,62
7504 DATA 1,50,102,204,201
7505 DATA 20,0,0,0,0
```

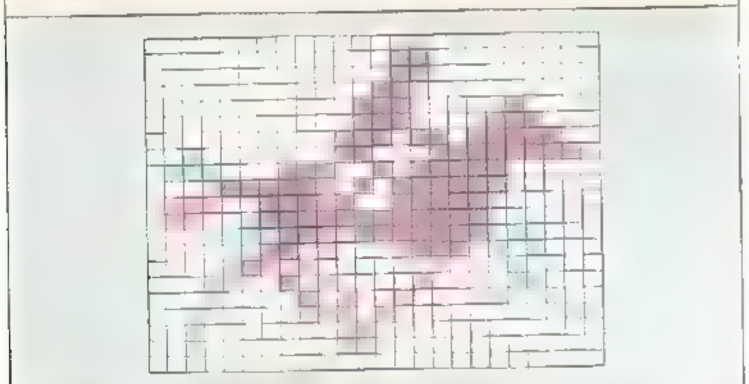
ANIMATION 1

So far the movement you have seen has been restricted to moving fixed sprites on the screen. This is all very well if your sprite is an aeroplane flying across the sky, or a car driving along, because these do not change shape as they move. However, if you want to have a moving person or a flying bird then something more sophisticated is required: the sprites need to be animated while they are moving on the screen.

Animation, like most techniques in computer graphics, relies on tricking the eye. It is achieved by displaying several stationary images in succession to create the illusion of movement. Animation can be achieved without any relative movement, as in an animated figure of a man jumping on the spot, for example. You will immediately notice when you start to animate your designs how only a small change in the sprite shape is required to give an effective result. By looking at the display in the figure above, for example, you will see how only the legs and tail of a horse need to be moved to give an animated effect; the horse's body remains unchanged. If the body was moved when the horse was animated, the effect would look very jerky.

ACHIEVING SMOOTH ANIMATION

The body of the horse sprite stays in the same position in successive frames (shown in red and blue); only the legs and tails of the horse are moved.

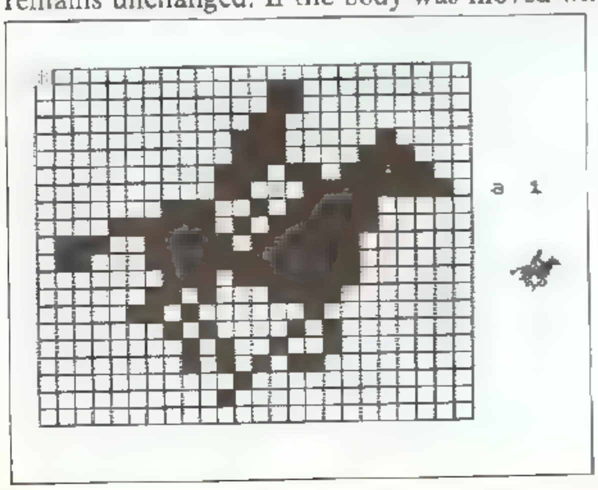


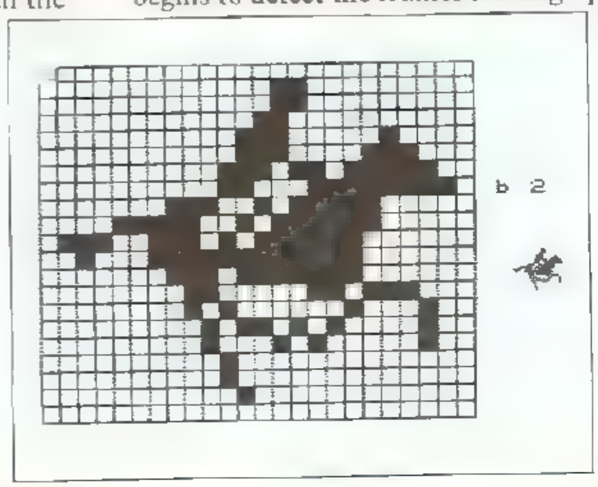
ANIMATION PROGRAM

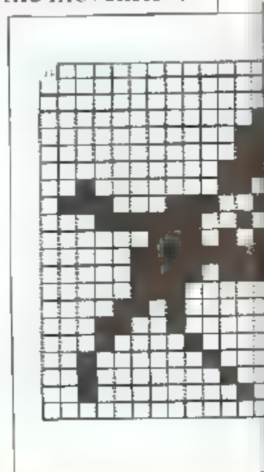
```
  10 DEF FN k(x,y,d,l,s,f,c,v,n)
=USR 61700
 100 BORDER 3: PAPER 5: INK 0: C
LS
 110 RANDOMIZE FN k(10,10,1,65,0
,5,0,50,1)
 120 RANDOMIZE FN k(205,10,0,65,
0,5,0,50,6)
 130 GO TO 100

0 OK, 0:1
```

In many cases you only need to take two frames and switch between them to produce convincing animation, but the animation routine, FNk, allows you to have as many frames as you like, within the limits of the sprite table – that is, up to a maximum of ten. Using ten frames, each of which differs slightly, you can create very effective illusions: unless you think about what is happening you would never realize that the sprites are being printed as stationary images.

The illusion produced by animation is also used in the cinema, where stationary images are shown one after the other at 14 frames per second. When watching a film you don't think of the picture as stationary, because the images are changing too fast for your eye to register. Since the animation routine allows you to vary the speed at which images are replaced on the screen, you can experiment by giving different values to the v parameter to see how slow the animation can be before your eye begins to detect the frames making up the movement.







FNk

SPRITE ANIMATION ROUTINE

Start address 51700 **Length** 275 bytes
Other routines called Master sprite routine.
What it does Uses a sequence of sprites from the sprite buffer to give the effect of animation.

Using the routine Most of the routine parameters are as before, with some additional ones added to give you increased control of the animated sprites. Thus, the frame parameter specifies how many frames are used in the animation, and the velocity parameter determines how quickly the routine should move from one frame to the next. Do not give this parameter a value of 0.

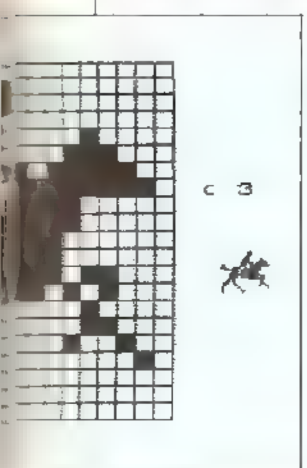
ROUTINE PARAMETERS	
DEF FNk(x,y,d,l,s,f,c,v,n)	
x,y	sprite start co-ordinate (0<x<256, 0<y<176)
d	specifies direction of travel (0=left, 1=right, 2=up, 3=down)
l	distance moved (vertical<=51, horizontal<=77)
s	switch (1=switch on, 0=switch off)
f	number of frames used in the animation (1-10)
c	flag for collision detection (1=on, 0=off)
v	velocity of animation (1<=v<=255, the slowest)
n	number of first sprite (1-10)

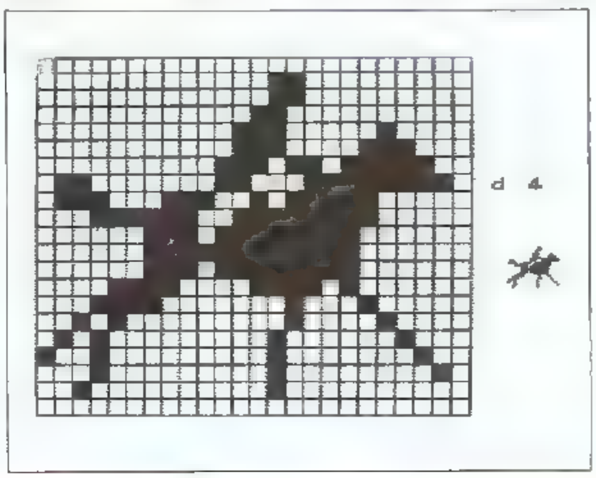
The animation program

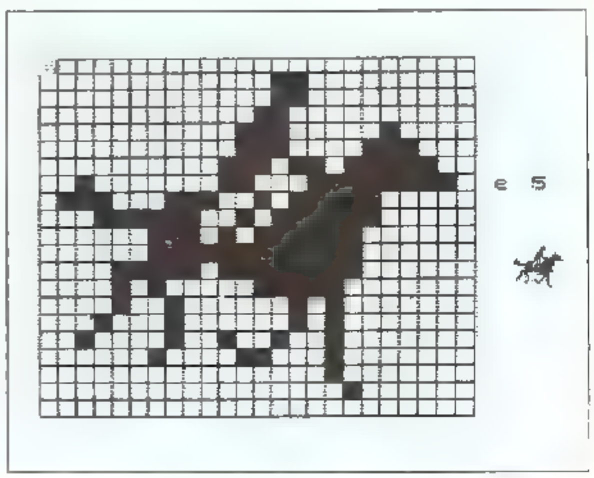
This program has not been displayed for the obvious reason that it is difficult to capture the effect of movement in a still photograph. The program takes five sprites and displays them in turn to give a very smooth effect of movement. The five sprites are shown along the bottom of the page as they appear on the sprite editor. Notice how similar each horse is to the next; only minor changes are needed for the animation. For an even better result, increase the number of frames.

ROUTINE LISTING

```
7350 LET b=51700: LET l=270: LET
 z=0: RESTORE 7360
7351 FOR i=0 TO l-1: READ a
7352 POKE (b+i),a: LET z=z+a
7353 NEXT i
7354 LET z=INT (((z/l)-INT (z/l)
)*l)
7355 READ a: IF a<>z THEN PRINT
"??": STOP

7360 DATA 42,11,92,17,4
7361 DATA 0,25,78,30,8
7362 DATA 25,70,25,126,230
7363 DATA 3,50,209,202,25
7364 DATA 126,50,211,202,25
7365 DATA 126,230,1,50,212
7366 DATA 202,25,126,50,216
7367 DATA 202,50,215,202,25
7368 DATA 126,230,1,50,210
7369 DATA 202,25,126,50,217

7370 DATA 202,25,126,17,63
7371 DATA 0,33,202,212,25
7372 DATA 61,32,252,34,213
7373 DATA 202,58,215,202,61
7374 DATA 50,215,202,254,0
7375 DATA 32,9,58,216,202
7376 DATA 50,215,202,42,213
7377 DATA 202,17,63,0,25
7378 DATA 205,93,210,254,0
7379 DATA 40,19,58,210,202

7380 DATA 254,0,40,12,58
7381 DATA 212,202,198,1,230
7382 DATA 1,50,212,202,24
7383 DATA 90,229,213,197,58
7384 DATA 217,202,71,17,0
7385 DATA 0,33,0,0,237
7386 DATA 176,193,209,22[illegible],118
7387 DATA 205,196,209,58,209
7388 DATA 202,254,0,40,19
7389 DATA 254,1,40,26,254

7390 DATA 2,40,33,5,5
7391 DATA 5,120,230,252,40
7392 DATA 45,195,182,202,13
7393 DATA 13,13,121,230,252
7394 DATA 40,34,195,182,202
7395 DATA 12,12,12,121,214
7396 DATA 231,48,23,195,18[illegible]
7397 DATA 202,4,4,4,120
7398 DATA 214,150,48,12,58
7399 DATA 211,202,61,50,211

7400 DATA 202,254,0,194,54
7401 DATA 202,58,212,202,237
7402 DATA 67,242,201,254,0
7403 DATA 200,118,205,93,210
7404 DATA 201,1,1,7,0
7405 DATA 9,213,1,3,40
7406 DATA 231,48,22,195,232
7407 DATA 202,4,4,4,120
7408 DATA 214,150,48,11,58
7409 DATA 3,203,61,50,3
7410 DATA 203,254,0,32,155
7411 DATA 58,4,203,237,67
7412 DATA 86,202,254,0,200
7413 DATA 118,205,93,210,201
7414 DATA 63,0,0,0,0
```







ANIMATION 2

How does the animation routine work? The answer is simple for you, but not quite so simple for the programmer. What happens with ordinary sprites is that they are picked out of memory, printed on the screen and moved across it, according to the parameters passed to the routine. With animation the first sprite is picked out of memory, displayed on the screen, and moved one position (three pixels in the case of the sprites in this book). The sprite is then deleted from the screen, and the next sprite in the sequence (the equivalent of the next frame in a film, or next drawing in an animated cartoon) receives exactly the same treatment. It is printed on the screen, moved three pixels and then wiped off again.

The sequence continues until all the frames in the sequence have been displayed, and then starts again. Some applications are ideally suited to two frames (like birds flying), but obviously the more frames in the sequence, the smoother the animation will be. For this reason it is best to choose to animate things which have a regular and repetitive movement.

Transferring animation to the screen

Sooner or later you will want to start designing your own animated characters. The technique recommended is to begin by reproducing the movement you see, as simply as possible. Remember that your eye will help by persuading you that things resemble real life, even though they are not. Secondly, remember that the effect you are aiming for is a flowing movement. To achieve this it is necessary to make sure that the last frame in the sequence runs into the first, so that the sequence is circular — after all this is the way it will be projected onto the screen. One of the commonest errors made in animation is to have an open-ended sequence, so that when it is shown repeatedly on the screen the effect is smooth during the sequence, but with a jerk at the end as the sequence restarts.

You are fortunate enough to have the sprite editor and the sprite print routine to help you with your designs. You will find that designing on the screen is much easier than sitting down and working out an image with a pencil and a grid. As a further aid, several animated sequences have been provided in the sprite design directory later in the book. Using the sprite editor, you can start a new sprite design from a previous one and so create smooth and flowing animation sequences with little difficulty.

Trying out the animation

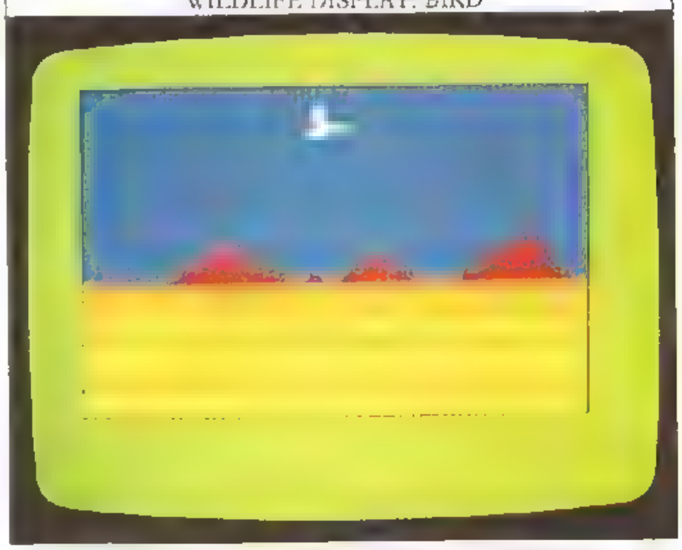
When you have designed an animation sequence you can use the sprite print routine to preview your designs. You saw this idea used with a robot earlier in the book, but the point of using the print routine here is twofold.

The first point is that it is much easier to be analytical and critical about a sprite design if it is not moving across the screen. If you print your sequence one after the other onto the same screen position in a continuous loop you will get some idea about where changes need to be made (if any) before the sequence is animated properly.

Secondly, you will be able to judge more accurately the sort of speed at which the sequence should be animated. Put a PAUSE statement between each print statement in the loop, starting with values of about 20. You will then be able to see how much the animation routine needs to be slowed down to be most effective. In practice you should multiply the results you get from tests with the PAUSE statements by a factor of between five and ten (that is, PAUSE 20 converts to a value for v of 100), simply to compensate for the fact that machine code is so much faster than BASIC.

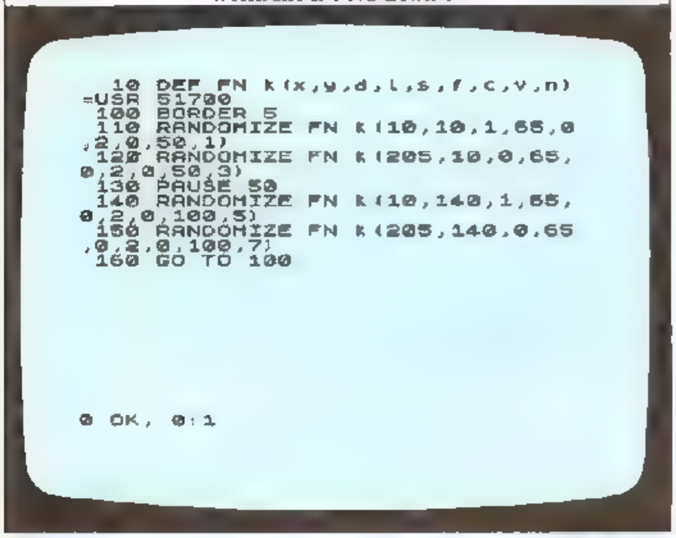
It is a good idea to make use of the sprite print routine for one more reason. Once a sprite is being animated with

WILDLIFE DISPLAY: BIRD



the animation routine it is impossible to stop it until the distance l has been covered. This can take quite a while if you have long pauses between each frame of the sequence. However, with the sprite print routine you can break into the program between single frames at any point since the Spectrum is returning to BASIC between each call to the print routine.

WILDLIFE PROGRAM

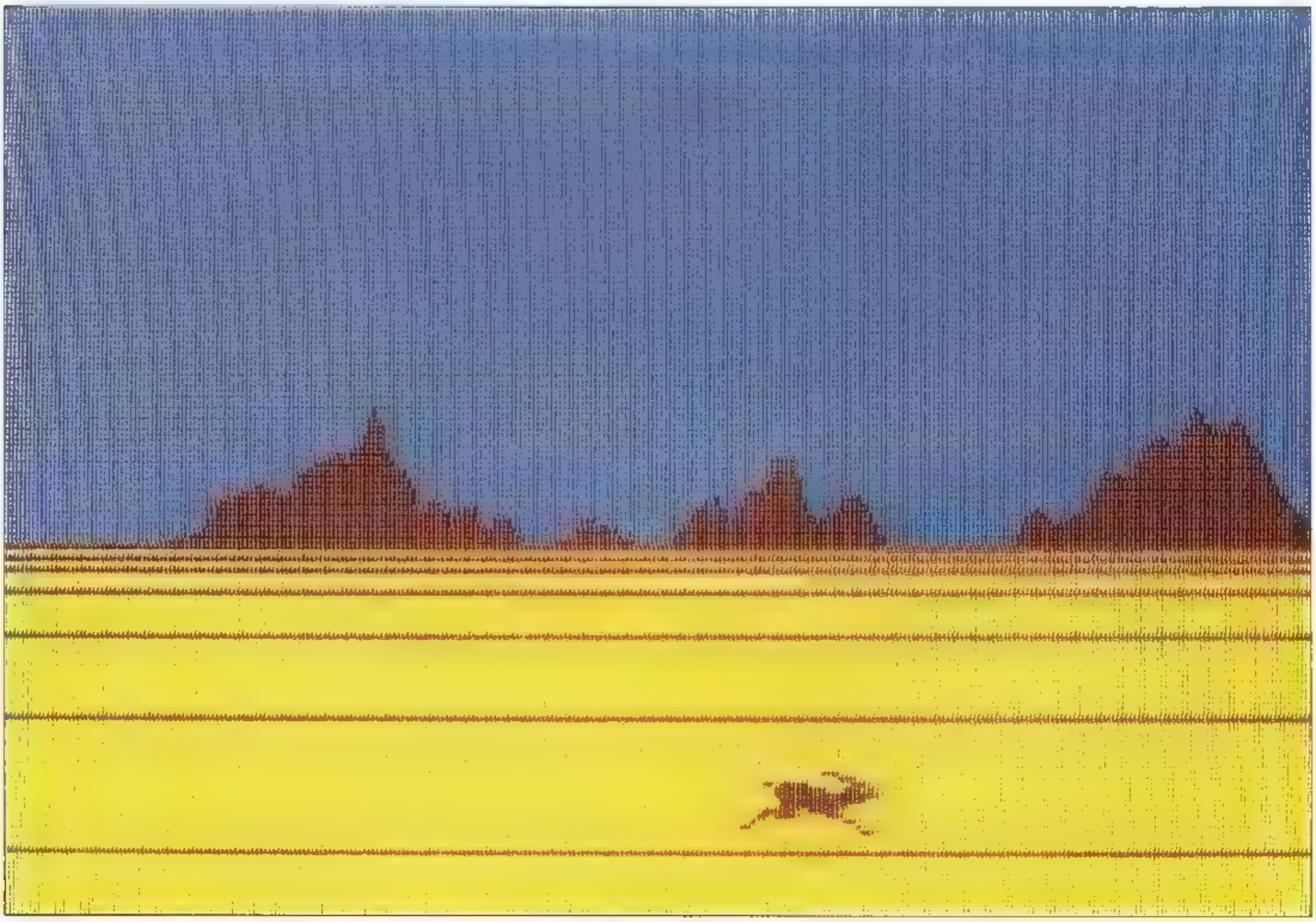


```
  10 DEF FN k(x,y,d,l,s,f,c,v,n)
=USR 51700
 100 BORDER 5
 110 RANDOMIZE FN k(10,10,1,65,0
,2,0,50,1)
 120 RANDOMIZE FN k(205,10,0,65,
0,2,0,50,3)
 130 PAUSE 50
 140 RANDOMIZE FN k(10,140,1,65,
0,2,0,100,5)
 150 RANDOMIZE FN k(205,140.0.65
,0,2,0,100,7)
 160 GO TO 100

0 OK, 0:1
```

Once you are completely satisfied with the edited results and the way that they work with the sprite print routine then you should be sure to save the sprites generated to tape, as a separate file. The reason for this is that after all the effort you have spent getting the sprite right it would be a pity to lose them through a clerical error — or a momentary power cut.

The wildlife program

The displays on this page are both produced by a single program. Both displays use two states of animation, and both move the animated sprite across the screen and back again. The movement of the hare (shown in the large display below) is simplified to two states — the elongated position, and the familiar crouching pose. Alternating between these two provides a reasonable approximation of movement. The two bird sprite states, shown in detail in the close-up photograph, show how both the wings and body of the bird are made to move. You may find the bird's movement rather jerky. One way of overcoming this, without increasing the number of frames, would be to reduce the displacement of the body by keeping it more central, rather than rising and falling within the 24 by 21 frame with each animation state.



SCREEN SCROLLING

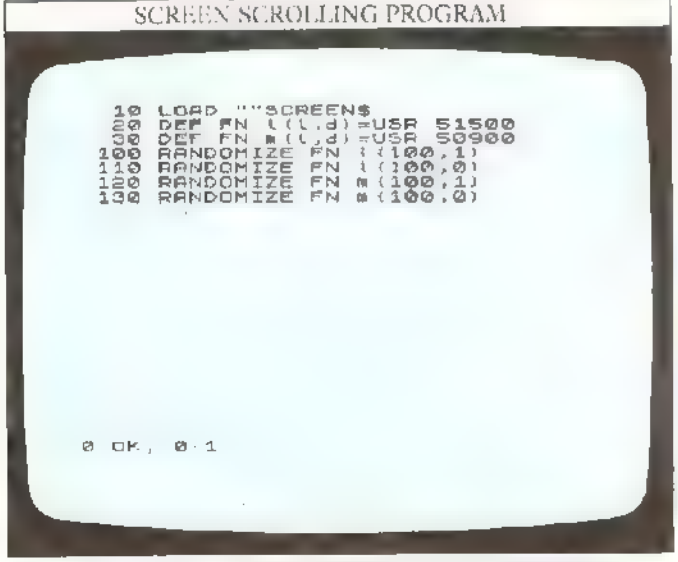
All the movement you have seen so far relies on the idea of you being stationary and something moving past you. But how can you create the illusion of moving past something which is stationary? The most effective way of doing this is by scrolling the whole screen. There are several ways of doing this; the simplest can be seen whenever you use the BASIC command LIST, which scrolls the screen upwards one character at a time.

The scroll routines

To create a more effective and gentle illusion of movement you need a smoother scroll, which moves the screen one pixel at a time. The two routines given here, FNl and FNm, allow you to scroll the screen a pixel at a time in either a horizontal or a vertical direction.

Note that when using the two scrolling routines on this page, it is inadvisable to use an interrupt-driven routine (such as the keyboard-controlled sprite routine.

SCREEN SCROLLING PROGRAM



SCROLLING THE SCREEN HORIZONTALLY

SCROLLING THE SCREEN VERTICALLY

FNh) at the same time. This is because the scrolling routines are shifting the co-ordinates of screen memory back and forth as the screen is scrolled, and an image is displayed on the screen by interrupts being used to "refresh" the screen at regular intervals. Obviously, this can cause problems if the screen display moves its location in memory when the screen refresh routine looks for it.

A similar difficulty arises when using PAUSE, since this statement is also based on interrupts. Rather than use PAUSE between two calls to a scroll routine, you are advised to use a FOR...NEXT loop for a delaying effect.

Other scrolling effects

Several effects can be linked to scrolling the screen. Both of the scroll routines have a wraparound effect, which means that whatever goes off one edge of the

FNl

HORIZONTAL SCROLL ROUTINE

Start address 51500 **Length** 190 bytes
What it does Scrolls the screen left or right a specified distance.

Using the routine Parameters d and l set the direction and length of scroll. The routine has a wraparound effect, so that whatever disappears off the left of the screen reappears on the right, and vice versa.

ROUTINE PARAMETERS

DEF FNl (l,d)	
l	specifies length of scroll (0–255)
d	specifies direction of scroll (0=left, 1=right)

ROUTINE LISTING

```
7300 LET b=51500: LET l=185: LET
 z=0: RESTORE 7310
7301 FOR i=0 TO l-1: READ a
7302 POKE (b+i),a: LET z=z+a
7303 NEXT i
7304 LET z=INT (((z/l)-INT (z/l)
)*l)
7305 READ a: IF a<>z THEN PRINT
"???": STOP

7310 DATA 243,237,115,141,201
7311 DATA 42,11,92,17,4
7312 DATA 0,25,70,30,8
7313 DATA 25,126,254,0,32
7314 DATA 33,14,191,197,33
7315 DATA 0,64,229,6,32
7316 DATA 167,203,30,35,16
7317 DATA 251,225,48,4,62
7318 DATA 128,182,119,17,32
7319 DATA 0,25,13,32,233

7320 DATA 193,16,224,237,123
7321 DATA 141,201,251,201,14
7322 DATA 191,197,33,31,64
7323 DATA 229,6,32,167,203
7324 DATA 22,43,16,251,225
7325 DATA 48,4,62,1,182
7326 DATA 119,17,32,0,25
7327 DATA 13,32,233,193,16
7328 DATA 224,201,237,123,141
7329 DATA 201,251,0,0,0

7330 DATA 42,11,92,17,4
7331 DATA 0,25,70,30,8
7332 DATA 25,126,254,0,32
7333 DATA 33,14,191,197,33
7334 DATA 0,64,229,6,32
7335 DATA 167,203,30,35,16
7336 DATA 251,225,48,4,62
7337 DATA 128,182,119,17,32
7338 DATA 0,25,13,32,233
7339 DATA 193,16,224,201,14

7340 DATA 191,197,33,31,64
7341 DATA 229,6,32,167,203
7342 DATA 22,43,16,251,225
7343 DATA 48,4,62,1,182
7344 DATA 119,17,32,0,25
7345 DATA 13,32,233,193,16
7346 DATA 224,201,0,0,0
7347 DATA 61,0,0,0,0
```

screen then reappears on the opposite edge. Other routines allow you to leave something stationary on the screen while scrolling the rest — as used in popular games like Defender and Pole Position. To do this requires a much longer routine than those given here.

However, another type of scrolling effect has been included in this book. This is ■ partial screen scroll, in which the vertical dimension of the area scrolled is

FNm

VERTICAL SCROLL ROUTINE

Start address 50900 **Length** 215 bytes
What it does Scrolls the screen ■ specified distance up or down.

Using the routine Parameters are the same as for the horizontal scroll routine. As before, a wraparound effect occurs with the routine, but in this case when scrolling off the top and bottom of the screen.

ROUTINE PARAMETERS

DEF FNm(l,d)	
l	specifies length of scroll (0–175)
d	specifies direction of scroll (0—up, 1—down)

ROUTINE LISTING

```
7200 LET b=50900: LET l=210: LET
 z=0: RESTORE 7210
7201 FOR i=0 TO l-1: READ a
7202 POKE (b+i),a: LET z=z+a
7203 NEXT i
7204 LET z=INT (((z/l)-INT (z/l)
)*l)
7205 READ a: IF a<>z THEN PRINT
"???": STOP

7210 DATA 243,42,11,92,17
7211 DATA 4,0,25,70,3■
7212 DATA 8,25,126,230,1
7213 DATA 40,91,197,1,32
7214 DATA 0,33,0,64,17
7215 DATA 162,199,126,18,35
7216 DATA 19,16,250,1,175
7217 DATA 0,33,0,64,17
7218 DATA 0,65,213,229,6
7219 DATA 32,26,119,35,19

7220 DATA 16,250,2■5,209,36
7221 DATA 62,7,164,32,10
7222 DATA 62,32,133,111,40
7223 DATA 4,124,214,8,103
7224 DATA 20,62,7,162,32
7225 DATA 10,62,32,131,95
7226 DATA 40,4,122,214,8
7227 DATA 87,13,32,209,6
7228 DATA 32,33,162,199,17
7229 ■AT■ 160,87,126,18,35

7230 DATA 19.16,250,193,16
7231 DATA 167,251,201,197,33
7232 DATA 160,87,17,162,199
7233 DATA 6,32,126,18,35
7234 DATA 19,16,250,1,175
7235 DATA 0,33,160,36,17
7236 DATA 160,87,213,229,6
7237 DATA 32,126,18,35,19
7238 DATA 16,250,225,209,37
7239 DATA 124,230,7,254,7

7240 DATA 32,12,125,214,32
7241 DATA 111,254,224,40,4
7242 DATA 62,8,132,103,21
7243 DATA 122,230,7,254,7
7244 DATA 32,12,123,214,32
7245 DATA 95,254,224,40,4
7246 DATA 62,8,130,87,13
7247 DATA 32,201,17,0,64
7248 DATA 33,162,199,6,32
7249 DATA 126,18,35,19,16
7250 DATA 250,193,16,160,251
7251 DATA 201,0,0,0,0
7252 DATA 47,0,0,0,0
```

restricted to the size of the sprites used in this book. This partial screen scroll routine (given on page 30) produces what is in effect a window.

The program on this page shows scrolling at work. You will notice that the routine has the effect of moving ink attributes while leaving coloured areas unchanged.

WINDOWS 1

As you saw on pages 28-29, it is often more useful to be able to scroll parts of the screen than to scroll all of it. The window routine given here, FNn, enables you to do this. With the routine, you can define an area of the screen three characters deep of any width, and then move a sprite across it. The routine enables you to make sprites appear to move behind objects on the screen, since they appear from outside the window and then disappear the other side — rather like looking out of a window and watching a train go past.

Repeating sprites

One additional feature of the window routine is that you can repeat the sprite to give the effect of a chain of sprites passing through the window. Alternatively, quite spectacular effects can be achieved by having a variety of sprites set up in the sprite table, like the sequence produced by the cockpit program shown on this page.

Differences between sprites and windows

In many cases, you may find it more useful in your programs to use this window routine than a standard sprite routine. However, when using the routine, it is important to remember that the routine is essentially a scrolling, rather than a sprite routine, and that everything contained in the window will be scrolled by the routine, even if it was only part of the original background. To display a sprite moving against a static background, or to move a sprite vertically, you must use one of the sprite routines.

COCKPIT PROGRAM

```
  10 DEF FN n(x,y,l,n,d,r)=USR 4
9600
 100 FOR x=1 TO 10
 110 IF x<>2 AND x<>5 AND x<>7 T
HEN RANDOMIZE FN n(18,8,13,x,1,1
): RANDOMIZE FN n(1,8,13,x,1,1):
 PAUSE 50
 120 IF x=2 OR x=5 OR x>=7 THEN
RANDOMIZE FN n(1,8,13,x,0,1): RA
NDOMIZE FN n(18,8,13,x,0,1): PAU
SE 50
 130 NEXT x
 140 GO TO 100

0 OK, 0:1
```

FNn

WINDOW ROUTINE

Start address 49600 **Length** 290 bytes
What it does Moves one or several sprites across a window of specified dimension.

Using the routine The routine carries out a partial screen scroll: everything contained within the area defined by the parameters is scrolled.
Note that the start co-ordinates x,y represent the top left hand corner of the sprite if the sprite moves right, but the top right-hand corner of the sprite if the sprite moves left. The repeat flag, r, can be used to repeat a sequence of sprites moving across the window. Repeated sprites can be seen in the window game program on pages 32-33.

ROUTINE PARAMETERS

DEF FNn(x,y,l,n,d,r)	
x,y	start co-ordinates (0<=x<=31, 0<=y<=21)
l	width of the window (0<=l<=31)
n	number of sprite to be scrolled (n=1-10)
d	specifies direction of scroll (0=right, 1=left)
r	repeat flag (1=switch off repeat, 0=repeat)

The cockpit program

The large display on this page shows windows at work. From the interior of an aircraft cockpit, various air- and spacecraft can be seen flying across in front of the plane; the display shows one of these. What is particularly effective is the way the sprites disappear behind the centre pillar and reappear in the other window — an effect which could not be achieved with the sprite routines. This program, then, shows the real difference between windows and sprites — using windows you can make the sprites look as if they are behind a solid object rather than in front of it.

How the program works

Line 10 of the program defines the window routine, together with its assortment of parameters. As you can see, three characters wide is a very effective width for the routine. The program actually uses two windows, three characters apart, with a barrier (the window frame) in the middle of the screen. The eye, however, is tricked into believing that the sprites are moving along behind the windows when they are in fact disappearing, stopping, and then reappearing as a result of a second window call. Line 100 sets up a loop so that all ten sprites can be made to appear across the window, one by one.

Line 120 deals with the sprites that are left facing — that is, all of the sprites which appear to fly from right to left. This works by testing for the second, fifth and seventh sprites which are right facing. If any other sprite is to be used then it is first made to go across the right-hand window, immediately followed by a call to make it go across the left-hand window. A PAUSE statement has been added to give you time to think about what just flew in front of your eyes!

Line 130 behaves in the same way, but handles the right facing sprites. Sprites 2, 5, and 7 and upwards appear to face to the right, so these are made to fly across the left-hand window first, immediately followed by a second call to make them fly across the right-hand one. Line 140 ends the loop, and finally line 150 starts the whole process over again.

ROUTINE LISTING

```
7100 LET b=49600: LET l=285: LET
 z=0: RESTORE 7110
7101 FOR i=0 TO l-1: READ a
7102 POKE (b+i),a: LET z=z+a
7103 NEXT i
7104 LET z=INT (((z/l)-INT (z/l)
)*l)
7105 READ a: IF a<>z THEN PRINT
"??": STOP

7110 DATA 42,11,92,1,4
7111 DATA 0,9,36,14,8
7112 DATA 9,94,9,126,50
7113 DATA 218,194,9,126,50
7114 DATA 223,194,9,126,230
7115 DATA 1,50,222,194,9
7116 DATA 126,230,1,50,221
7117 DATA 194,62,1,50,219
7118 DATA 194,50,224,194,123
7119 DATA 230,24,246,64,103

7120 DATA 123,230,7,183,31
7121 DATA 31,31,31,130,111
7122 DATA 58,222,194,254,0
7123 DATA 40,6,58,218,194
7124 DATA 133,61,111,58,218
7125 DATA 194,60,60,60,71
7126 DATA 197,229,205,102,194
7127 DATA 225,6,8,229,197
7128 DATA 17,225,194,58,222
7129 DATA 194,254,0,40,3

7130 DATA 17,11,196,6,3
7131 DATA 229,197,120,254,1
7132 DATA 32,4,6,5,24
7133 DATA 2,6,8,197,213
7134 DATA 229,58,222,194,254
7135 DATA 0,32,5,205,160
7136 DATA 194,24,3,205,189
7137 DATA 194,225,209,193,36
7138 DATA 19,16,231,193,225
7139 DATA 62,32,133,111,48

7140 DATA 4,62,8,132,103
7141 DATA 16,204,193,225,16
7142 DATA 183,193,118,16,171
7143 DATA 201,58,224,194,61
7144 DATA 50,224,194,192,62
7145 DATA 3,50,224,194,58
7146 DATA 223,194,17,63,0
7147 DATA 71,33,72,213,167
7148 DATA 237,82,25,16,253
7149 DATA 67,17,225,194,58

7150 DATA 219,194,254,0,4[illegible]
7151 DATA 1,126,18,35,19
7152 DATA 16,243,58,221,194
7153 DATA 254,0,200,62,0
7154 DATA 50,219,194,201,1[illegible]7
7155 DATA 6,3,213,26,31
7156 DATA 18,245,62,21,131
7157 DATA 95,48,1,20,241
7158 DATA 16,242,209,58,218
7159 DATA 194,71,203,30,35

7160 DATA 16,251,201,167,213
7161 DATA 6,3,26,23,18
7162 DATA 245,123,214,21,95
7163 DATA 48,1,21,241,16
7164 DATA 242,209,58,218,194
7165 DATA 71,203,22,43,16
7166 DATA 251,201,10,0,0
7167 DATA 43,0,0,0,0
```

WINDOWS 2

The window routine given on this page has all the features of the window routine, FNn, but with the added feature of being interrupt-driven. Once switched on, it will keep going until switched off again, regardless of just about any BASIC command.

Your first priority with this routine must be to have a way of switching it off. As for keyboard-controlled sprites, this is done by a subroutine:

```
2000 DEF FN z(s)=USR 53100
2010 DEF FN x(s)=USR 49200
2020 RANDOMIZE FN z(1)
2030 RANDOMIZE FN z(0)
2040 RANDOMIZE FN x(0)
2050 RETURN
```

This subroutine is slightly more complicated than you would expect, since it is used both to set up the routine and to switch it off again. The routine also switches on the keyboard-controlled sprite routine, since this sets up the interrupt vector table required for the interrupt-driven window routine to work correctly.

WINDOW GAME PROGRAM

```
  10 DEF FN h(s,x,y,c,n)=USR 53100
  20 DEF FN g(x,y,d,l,s,c,n)=USR 53500
  30 DEF FN o(s,x,y,l,n,d,r)=USR 49200
  40 DEF FN z(s)=USR 49200
  50 DEF FN y(s)=USR 53100
 100 BORDER 1: PAPER 7: CLS
 110 LET sc=0: LET psc=0: LET g=-1: LET ng=0
 120 FOR i=1 TO 21
 130 PRINT INK 6;AT i,0;"██"
 140 PRINT INK 4,AT i,28;"██"
 150 PRINT INK 2,AT 1,i,"██"
 160 NEXT i
 170 PRINT INK 1,AT 5,4;"█ ██ ██ ██ █"
 180 GO SUB 500
 190 RANDOMIZE FN o(1,4,2,24,3,0,0)

scroll?
```

```
 200 RANDOMIZE FN h(1,32,150,0,10)
 210 IF INKEY$=" " THEN GO TO 230
 220 GO TO 210
 230 LET xp=PEEK 53098
 240 LET yp=PEEK 53099
 250 IF xp>=32 AND xp<=44 THEN LET sc=sc+2: LET ng=1
 260 IF xp>=80 AND xp<=92 THEN LET sc=sc+2: LET ng=2
 270 IF xp>=142 AND xp<=152 THEN LET sc=sc+2: LET ng=3
 280 IF xp>=190 AND xp<=200 THEN LET sc=sc+2: LET ng=4
 290 LET sc=sc-1
 300 IF sc+1=psc THEN LET ng=g+1
 310 IF ng=g THEN LET sc=sc-1: GO TO 340
 320 RANDOMIZE FN g(xp,yp-20,3,INT ((yp/3)),0,1,1)
 330 IF INKEY$<>"" THEN GO TO 33

scroll?
```

WINDOW GAME PROGRAM CONTD.

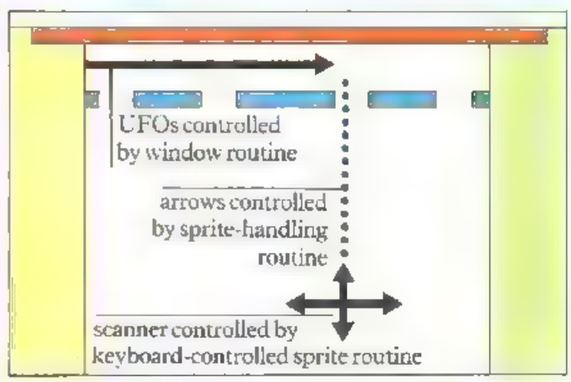
```
0
 340 IF sc<0 THEN LET sc=0
 350 LET g=ng: LET psc=sc
 360 GO SUB 500
 370 GO TO 210
 380 STOP
 500 PRINT PAPER 6; INK 0;AT 0,15;sc;
 510 RETURN
1000 RANDOMIZE FN y(1)
1010 RANDOMIZE FN y(0)
1020 RANDOMIZE FN z(0)



0 OK, 0:1
```

The interrupt-driven routine can be used together with the other routines in this book to set up and run sophisticated games. A simple game is given here; you will be able to improve it with a little effort.

The window game

This game is based on the idea of shooting at moving objects at the top of the screen. The game uses three machine-code routines. An interrupt-driven window is used to make flying saucers scroll across the top of the screen, the keyboard-controlled sprite routine is used to control your scanner, and the sprite-handling routine is used to send arrows towards the saucers.

WINDOW GAME SPRITE MOVEMENT



One of the best parts about writing a computer game is in deciding the scenario for the game. In this case you could imagine, for example, you are trapped on earth with only ■ scanner satellite, and your crew aboard it

FNo

INTERRUPT-DRIVEN WINDOW ROUTINE

Start address 49200 **Length** 315 bytes
Other routines called Keyboard-controlled sprite routine (FNh).
What it does Moves either one or several sprites across a window, and continues to operate whatever is happening in BASIC.

Using the routine To use this routine you must first switch on the keyboard routine, since this routine sets up tables of interrupts where needed. The keyboard sprite routine can then be switched off again, unless it is called in the program elsewhere. Use s in the same way as it is used in the keyboard controlled sprite routine.
Note that the start co-ordinates x,y of the window routine represent the top left-hand corner of the sprite if the sprite moves right and the top right-hand corner if it moves left.

ROUTINE PARAMETERS

DEF FNo(s,x,y,l,n,d,r)	
s	stop flag (1=stop, 0=normal)
x,y	start co-ordinates (0<=x<=31, 0<=y<=21)
l	width of window (0<=l<=31)
n	number of sprite to be scrolled (n=1-10)
d	specifies direction of scroll (0=right, 1=left)
r	repeat flag (1=switch off repeat, 0=repeat)

ROUTINE LISTING

```
7000 LET b=49200  LET l=310  LET
 z=0  RESTORE 7010
7001 FOR i=0 TO l-1  READ a
7002 POKE (b+i),a  LET z=z+a
7003 NEXT i
7004 LET z=INT (((z/l)-INT (z/l)
)*l)
7005 READ a  IF a<>z THEN PRINT
"???"  STOP

7010 DATA 243,42,11,92,1
7011 DATA 4,0,9,126,254
7012 DATA 0,32,9,17,252
7013 DATA 207,237,83,250,207
7014 DATA 251,201,17,139,192
7015 DATA 237,83,250,207,14
7016 DATA 3,9,36,9,94
7017 DATA 9,126,50,100,193
7018 DATA 9,126,50,108,193
7019 DATA 9,126,230,1,50

7020 DATA 107,193,9,126,230
7021 DATA 1,50,106,193,62
7022 DATA 1,50,104,193,50
7023 DATA 109,193,50,101,193
7024 DATA 123,230,24,246,64
7025 DATA 103,123,230,7,183
7026 DATA 31,31,31,31,130
7027 DATA 111,34,102,193,251
7028 DATA 201,58,107,193,254
7029 DATA 0,40,6,58,100

7030 DATA 193,133,61,111,58
7031 DATA 101,193,71,16,5
7032 DATA [illegible]05,240,192,6,8
7033 DATA 120,50,101,193,17
7034 DATA 110,193,42,102,193
7035 DATA 58,107,193,254,0
7036 DATA 40,3,17,152,193
7037 DATA 6,3,229,197,120
7038 DATA 254,1,32,4,6
7039 DATA 5,24,2,6,8

7040 DATA 197,213,229,58,107
7041 DATA 193,254,0,32,5
7042 DATA 205,42,193,24,3
7043 DATA 205,71,193,225,209
7044 DATA 193,36,19,16,231
7045 DATA 193,225,62,32,133
7046 DATA 111,48,4,62,8
7047 DATA 132,103,16,204,195
7048 DATA 252,207,58,109,193
7049 DATA 61,50,109,193,192

7050 DATA 62,3,50,109,193
7051 DATA 58,108,193,17,63
7052 DATA 0,71,33,72,213
7053 DATA 167,237,82,25,16
7054 DATA 253,67,17,110,193
7055 DATA 58,104,193,254,0
7056 DATA 40,1,136,18,35
7057 DATA 19,16,243,58,106
7058 DATA 193,254,0,200,62
7059 DATA 0,50,104,193,201

7060 DATA 167,6,3,213,26
7061 DATA 31,18,245,62,21
7062 DATA 131,95,48,1,20
7063 DATA 241,16,242,209,56
7064 DATA 100,193,71,203,30
7065 DATA 35,16,251,201,167
7066 DATA 213,6,3,26,23
7067 DATA 18,245,123,214,21
7068 DATA 95,48,1,21,241
7069 DATA 16,242,209,58,100
7070 DATA 193,71,203,22,43
7071 DATA 16,251,201,5,4
7072 DATA 154,0,0,0,0
```

have only a bow and arrow with which to repel invaders. You can also use other sprites with the same background to change the feel of the game. Using different sprites, you could convert this game to a fairground shooting gallery, for example.

Several refinements could be made to the program. One sensible one would be to stop the keyboard sprite from going too high up the screen. This could be done by testing the current y co-ordinate of the sprite (stored at location 53098), and starting the sprite again at the bottom of the screen if a given limit is exceeded.

How the program works

Lines 10-20 define the three machine-code routines. Two of these routines are interrupt-driven and will therefore have to be switched off at some point. Lines 40 and 50 are here for this reason. These extra function definitions enable you to switch the interrupt-driven routines off and on by calling the routine with just one parameter, without bothering about specifying all the other parameters normally required.

Line 110 sets up some variables. The current score is held as variable sc, and the previous score (before the current arrow was fired) as psc. Variables g and ng are used to record the position of the keyboard sprite under the gaps, with ng defining the gap under which the sprite is resting, and g the previous gap. These variables can have values between 1 and 4 (since there are four gaps through which you can fire), and these variables are used to restrict your firing. The program compares variables g and ng in line 310, and if they are the same, the fire sequence is bypassed. At the beginning of the game, it is important that g and ng have different values, so that the fire sequence is not disallowed initially. Thus, g is set at first to -1, a value which ng can never have.

WINDOWS 3

Lines 120 to 170 of the program draw the background, with a loop used to create the sides and top, and line 170 creating the gaps through which you can fire. Line 180 calls the score subroutine at line 500.

Line 190 starts off the interrupt-driven window, though it does not begin working until the interrupt vector table has been set up by the keyboard-controlled sprite routine. Because this is the very next statement you do not see any delay. but it is important to remember that the interrupt-driven window will not function unless this table is in place. The window routine will continue to scroll a sprite across the window until something occurs to stop it.

Lines 210-220 form the main program loop, which simply waits for the space key (the signal to fire) to be pressed, since everything else at this stage is interrupt-controlled. The controlling keys for the keyboard sprite must of course be ignored, as their function is handled by the machine-code routine.

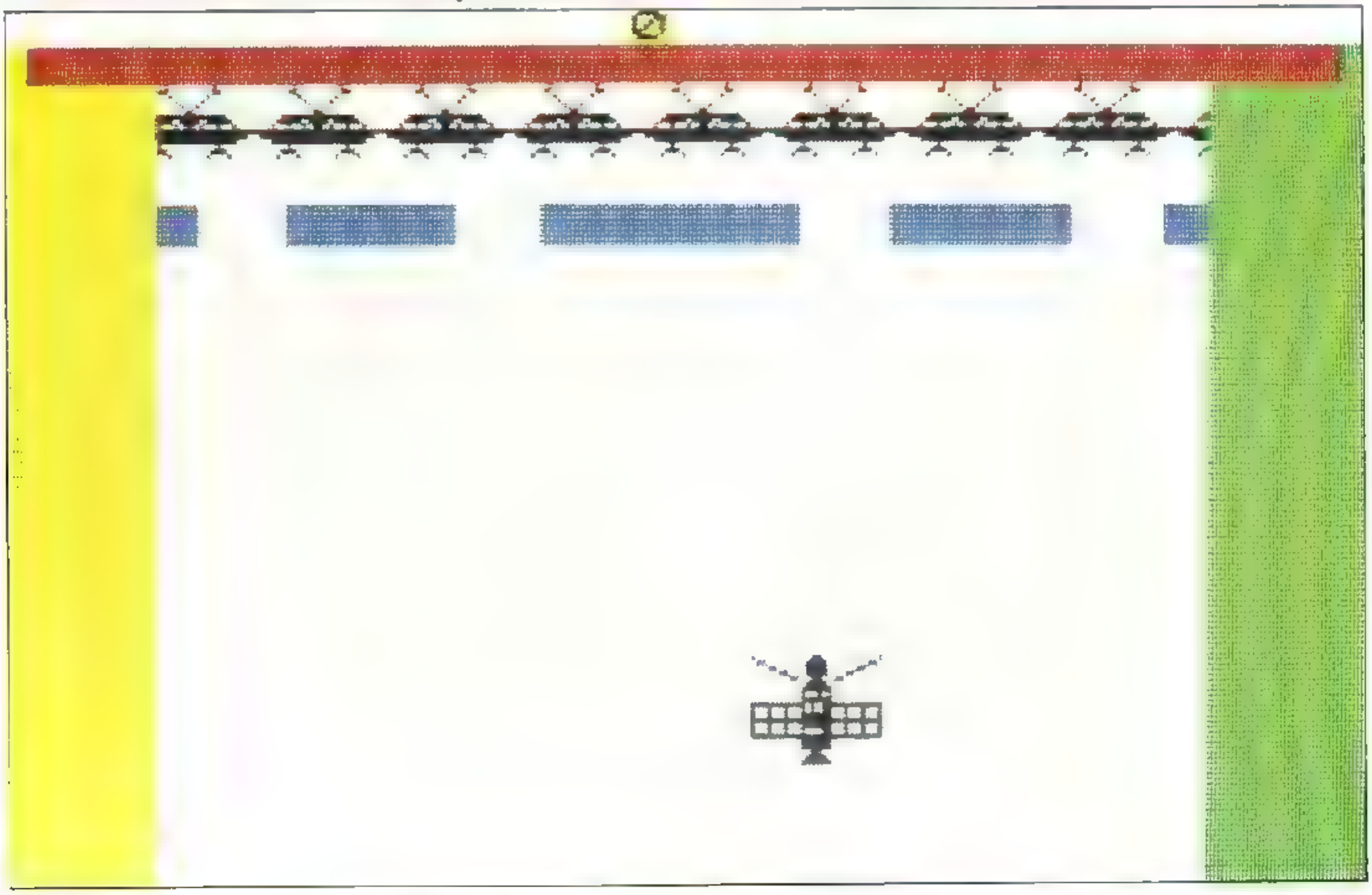
Lines 230 and 240 find out the current position of the keyboard-controlled sprite. Given the position of the sprite, and the gaps in the barrier, the program can decide whether a shot fired would go through a gap.

Lines 250-280 work in the same way for each of the four gaps. First a test is made to see whether the keyboard sprite is under the current gap. If it is then the gap flag (ng) is set to the value of the current gap. In addition the current score is incremented by two — not one as you might expect. Line 290 normalizes the score by taking one away again. Note that if the keyboard sprite is not under one of the gaps then the score is now one less than it was before the shot was fired.

Line 300 looks at the past score compared with the present score plus 1, and makes the gap variables different from their previous values, if the sprite was not under a gap. Line 310 deducts another one from the score if the gap is the same as the last gap which was fired at. It also skips the arrow-firing sequence, since arrows are only fired if the gap is a new one.

Line 280 fires an arrow by calling the sprite routine. Line 340 makes sure that the score cannot become negative, however bad the player. Line 350 adjusts the values of the previous score and the old gap value, ready for the next time round the loop. Line 360 prints the new score.

Finally, lines 1000-1020 switch off the interrupt-driven routines. To break out of the program, key GO TO 1000 to stop the routines.



USING THE SPRITE DIRECTORY

While using this book, you will have found that producing good sprite designs is not always easy. To overcome this problem, you can turn to the sprite directory, which contains over 200 sprite designs. These sprites can either be copied directly in your own programs, or used as a model on which to base your own ideas. Each entry in the directory includes the DATA for the sprite and shows the sprite on the screen.

Keying in the sprites

A sprite from the directory can be keyed in by using the following loading program (add DATA from line 100):

```
10 INPUT "Sprite number (1-10)",a
20 IF a<0 OR a>10 THEN GO TO 10
30 LET b=54600+((a-1)*63)
40 FOR i=b TO b+20
50 READ n : POKE i,n
60 READ n : POKE i+21,n
70 READ n : POKE i+42,n
80 NEXT i
100 DATA 0,0,0,0,etc
```

The routine loads the sprite in the DATA statements into the sprite location specified by you at the start of this BASIC program. The routine requires you to key in all 63 sprite DATA items, even if some of them are zero. The sprite table can easily be corrupted by wrong DATA being entered, but the easiest way to correct this is by editing the sprite image with the sprite editor.

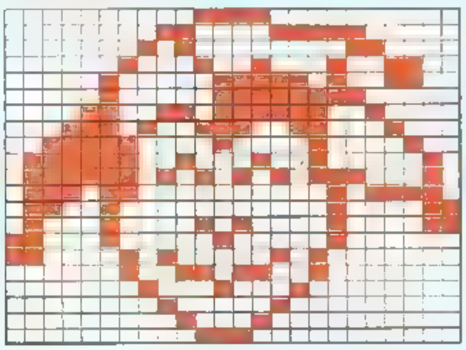
What the directory contains

The directory groups various kinds of sprite under theme headings. Most of the sprites are designed to be used individually, but the directory also contains some double sprites, which are used in conjunction with each other. In addition, a number of animation sequences of sprites in two or three different positions are included. The sprites can be entered either by keying in the DATA numbers provided, or simply by copying the drawing using the sprite editor. The sprite editor can also be used to increase the number of frames in an animation sequence, up to a maximum of ten designs (the maximum number of sprites that can be held in the buffer at one time).

EXAMPLES OF SPRITES

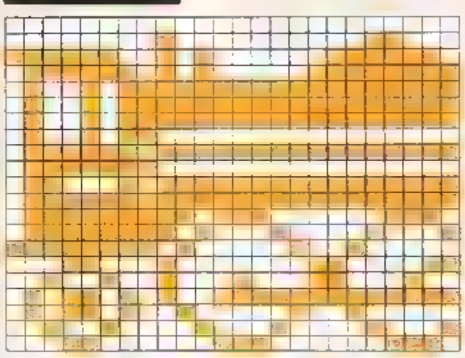
JOKER

```
0,60,0,0,195,224,1
64,48,2,26,76,4,31
76,4,31,224,12,223,144
28,57,136,62,16,132,126
68,130,125,170,226,118,0
94,100,0,67,68,40,67
196,16,64,194,120,128,2
108,128,1,17,0,0,130
0,0,68,0,0,56,0
```



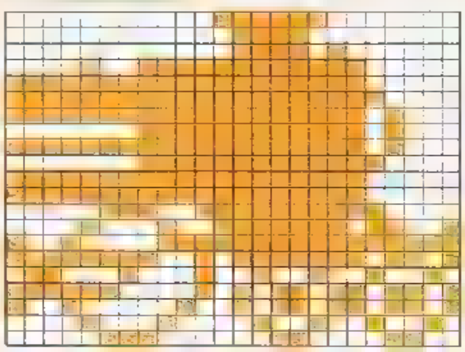
PACIFIC-TYPE LOCO

```
0,0,0,0,128,24,252
160,126,126,160,255,75,255
255,75,255,255,75,255,255
75,0,0,126,255,255,127
0,0,98,255,255,127,255
255,127,220,14,127,162,17
255,65,32,24,128,195,36
136,196,90,159,255,90,65
32,36,34,17,24,28,14
```



PACIFIC-TYPE LOCO

```
0,31,128,0,15,0,0
15,64,9,255,224,255,255
224,255,255,240,255,255,232
1,255,232,255,255,232,1
255,240,255,255,224,255,255
224,7,63,240,9,159,217
144,255,255,255,47,237,98
36,146,252,43,109,144,75
109,8,132,146,7,3,12
```



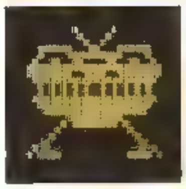
HOW SPRITES ARE SHOWN

The diagrams shown illustrate how sprites are displayed in the directory. Each sprite is shown in the top left-hand corner as it appears on the screen, and in the large display below as it can be keyed in on a grid. The 63 DATA numbers which are POKEd into memory are shown on the top right of each sprite display.

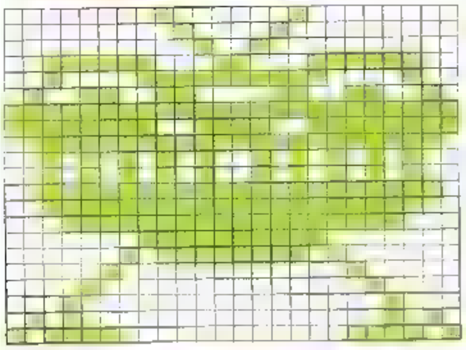
Note that double sprites are shown as two separate sprites; obviously, the sprites will appear joined when used with the double horizontal sprite routine (FNi).

ALIENS

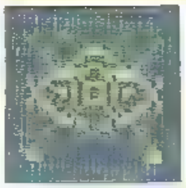
BUG



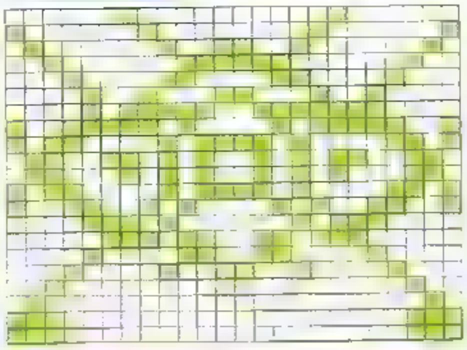
0,129,0,0,66,0,0
36,0,63,24,252,13,255
112,76,195,50,255,255,255
255,60,255,127,255,254,42
165,84,42,165,84,127,255
254,63,255,252,31,255,248
1,255,129,2,255,64,4
0,32,8,0,16,16,0
8,80,0,10,168,0,61



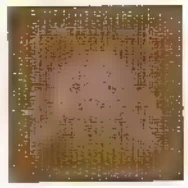
BUG



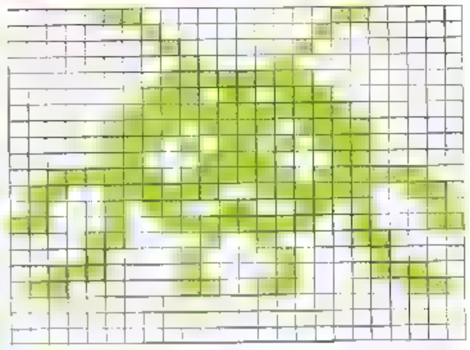
2,0,64,129,0,129,64
153,2,32,126,4,16,195
8,9,24,144,15,231,240
129,66,249,[illegible],189,[illegible],102
165,102,98,255,70,[illegible],165
13,153,66,153,14,129,112
7,132,224,[illegible],16,16
153,8,32,0,4,64,0
2,224,[illegible],224,0,7



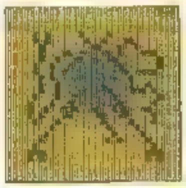
MICRO-MITE



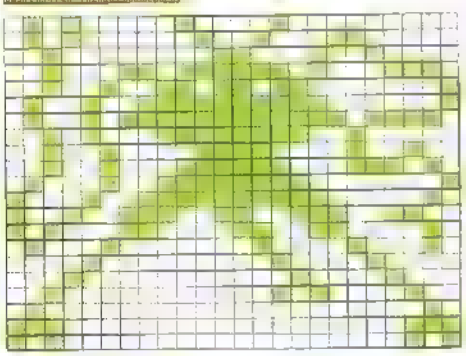
6,0,96,[illegible],0,128,0
129,0,0,66,0,0,255
0,1,219,128,3,255,192
7,189,224,7,90,224,6
60,96,63,126,252,103,231
230,69,167,162,196,255,35
12,36,49,8,66,16,56
66,28,96,102,6,0,0
0,0,0,0,0,0,0



TRIPOD

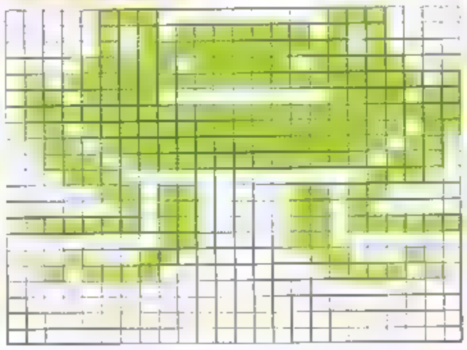


80,66,0,80,36,96,32
24,144,[illegible],128,142,35,219
1,73,255,129,15,255,127
40,255,32,36,60,36,36
126,58,68,255,1,137,255
130,147,151,204,163,12,194
63,4,34,8,2,16,16
1,8,32,2,132,64,0
2,224,0,7,160,0,5

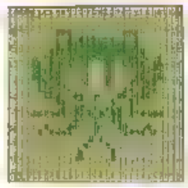


HOPPER

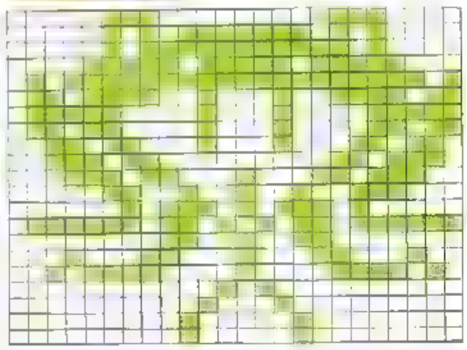
[illegible],12,[illegible]
[illegible],15,[illegible]
[illegible]
[illegible]
255,[illegible],58,0,58,6,[illegible]
[illegible],191
0,[illegible],128,[illegible],47
0,[illegible],0,0,0,0
0,0,0,0,0,0,0



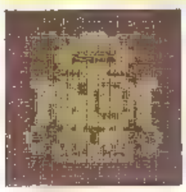
HOPPER



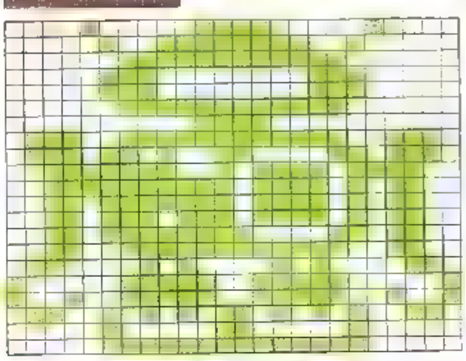
7,0,122,[illegible],124,[illegible]
255,208,15,191,240,15,255
[illegible],111,34,173,124,34,31
[illegible],34,2[illegible],34,[illegible],50
0,110,28,200,229,0,73
48,2,235,260,176,213,191
[illegible],201,128,[illegible],136,196,47
8,122,0,20,[illegible],0,14
0,0,65,0,0,65,0



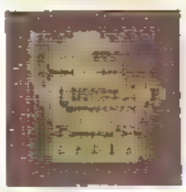
ROBOT



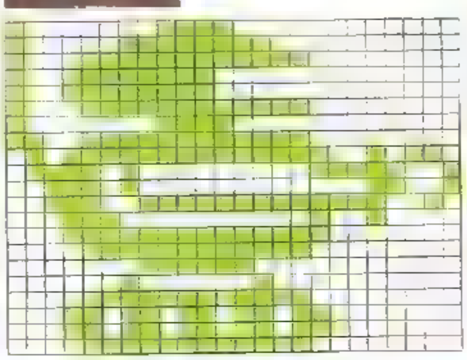
3,60,16,4,255,32,3
255,192,7,0,224,7,0
224,3,255,192,0,60,0
115,255,206,118,152,110,63
151,188,63,247,188,55,247
172,55,119,172,55,248,103
51,255,204,97,66,134,131
231,193,147,231,201,101,90
166,4,24,32,7,255,224



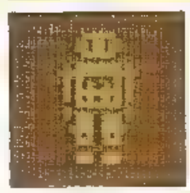
ROBOT



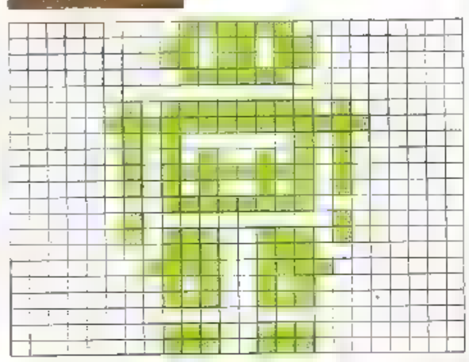
128,240,0,131,252,0,143
255,0,159,224,5,159,224
0,141,255,0,128,240,0
247,255,0,55,255,150,127
0,249,58,0,25,57,255
240,28,0,15,31,255,240
[illegible],255,224,1,254,0,2
1,0,15,255,224,27,109
176,27,159,176,15,255,224



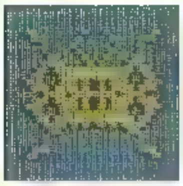
ROBOT



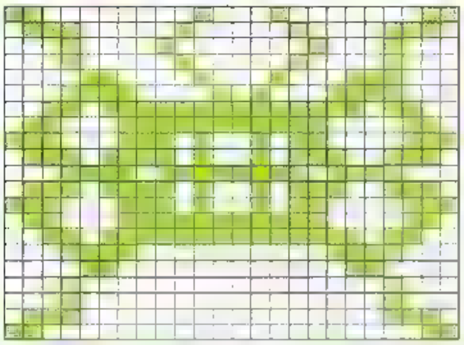
[illegible],126,0,0,219,0,0
219,0,0,126,0,0,0
0,2,255,64,7,255,224
2,129,64,2,165,64,2
255,64,2,165,64,0,255
0,2,0,64,2,231,64
0,231,0,0,231,129,0
165,0,0,231,0,0,0
0,0,231,0,0,231,0



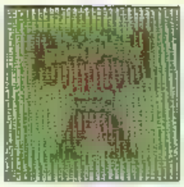
SQUAROID



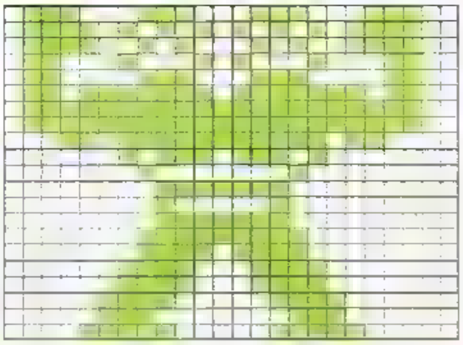
192,34,3,112,65,14,16
128,136,16,65,8,12,34
24,30,20,56,55,255,236
99,255,198,227,165,199,119
165,238,30,255,120,119,165
238,227,165,199,99,255,198
55,255,236,30,165,120,12
0,48,16,0,8,16,0
8,112,0,14,192,0,3



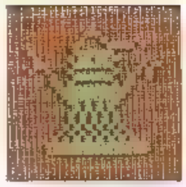
HUMANOID



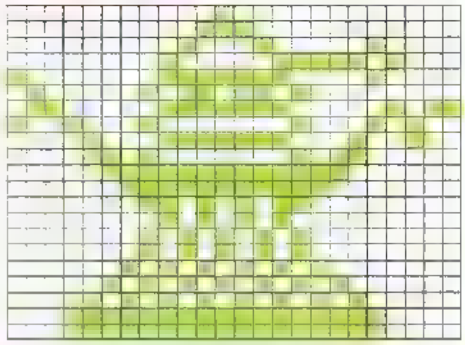
113,199,28,122,170,128,121
69,63,38,40,140,97,199
12,111,239,235,127,255,252
127,255,252,27,255,176,2
238,129,1,1,0,0,254
0,0,130,0,0,254,0
1,255,0,1,239,0,3
199,128,3,199,128,7,131
192,7,1,192,15,131,224



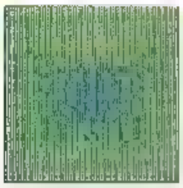
"DALEK"



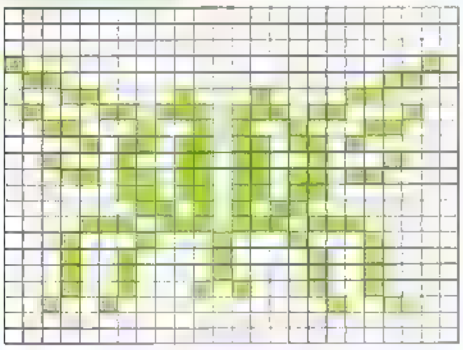
0,16,0,0,56,0,0
68,16,0,131,232,128,254
16,65,109,0,96,254,11
145,1,20,8,254,36,13
1,96,7,255,192,3,255
128,1,171,0,0,170,0
1,85,0,1,85,0,2
170,128,5,85,64,10,170
160,31,255,248,31,255,248



SPACE-FLY

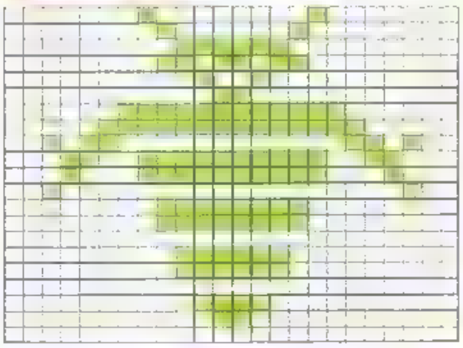


0,0,0,0,0,0,0
0,0,128,0,2,96,0
12,24,68,43,70,238,196
47,41,24,11,109,160,34
108,136,27,105,176,0,109
128,1,171,0,14,238,224
17,85,16,15,16,144,18
16,144,18,40,144,36,0
72,0,0,0,0,0,0



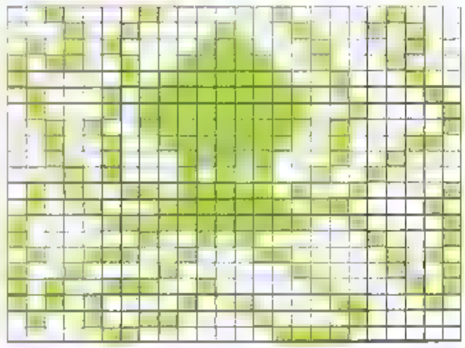
INSECTOID

1,0,128,0,129,0,0
126,0,7,219,0,0,36
0,0,24,0,3,255,192
7,25,,224,44,0,,2,25
255,152,17,255,136,32,0
4,0,255,7,7,255,0
0,0,0,0,126,0,0
126,0,0,,,0,0,60
0,0,24,0,0,0,0

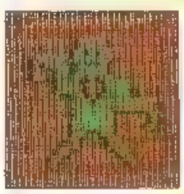


SEA MONSTER

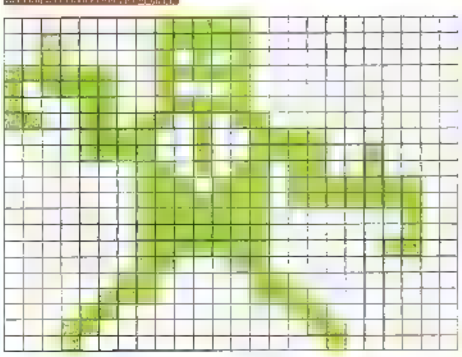
33,18,8,18,57,20,84
124,132,178,254,126,130,254
69,69,255,66,69,255,33
41,255,66,36,254,70,36
124,72,20,64,144,63,125
32,72,254,192,135,255,1
129,40,194,78,70,33,144
129,17,167,32,138,168,72
195,73,132,37,6,3,194



ANDROID

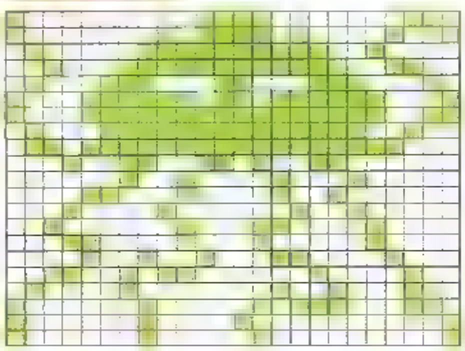


0,248,0,32,248,1,33
172,0,251,252,0,158,109
0,140,243,0,205,172,0
15,39,120,15,39,144,1
173,104,1,221,252,1,253
252,1,252,4,1,252,4
1,252,12,1,140,0,3
6,0,7,7,0,13,1
178,24,0,192,16,0,64

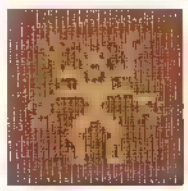


JELLY MONSTER

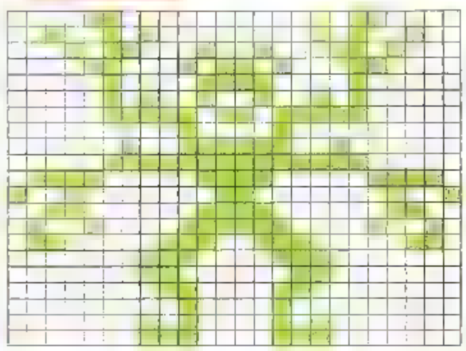
128,60,6,128,189,9,65
255,144,34,255,204,71,24
224,143,219,241,143,255,243
71,255,225,1,1,255,248,19
52,132,2,66,32,12,46
64,16,139,32,72,70,16
24,65,16,9,35,8,50
192,132,56,3,60,129,5
193,129,3,1,129,4,1



HYDRA MAN



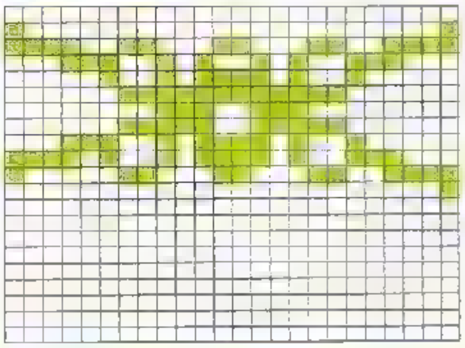
12,0,48,37,0,164,22
36,104,12,90,48,4,60
32,5,36,160,7,219,224
0,66,0,2,126,64,7
255,224,120,60,50,144,24
9,48,60,12,72,126,18
96,231,6,1,195,128,1
125,128,0,195,0,0,66
0,1,66,128,1,195,128



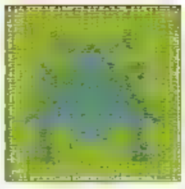
SPACECRAFT

PLANETARY PROBE

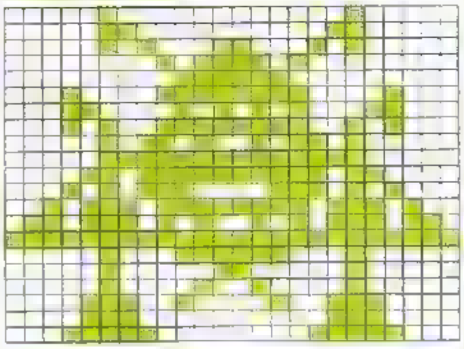
```
0,0,0,128,0,1,243
24,207,28,165,56,4,189
32,3,255,192,0,2[illegible]
[illegible],231,[illegible],189,32,28
189,56,243,24,207,128,0
1,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0
```



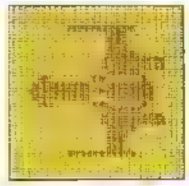
LANDER



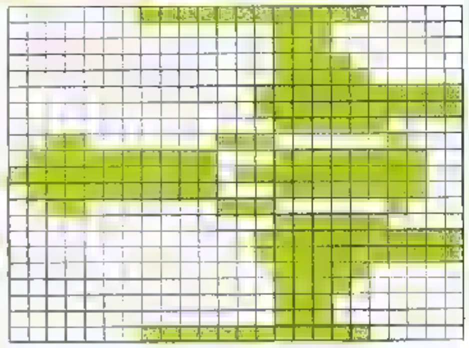
```
4,0,32,6,0,96,5
24,[illegible]
[illegible]
[illegible],255,[illegible],163,[illegible]
255,224,12,255,[illegible]
[illegible]
[illegible]
[illegible],56,[illegible],56,129
[illegible],56,[illegible],28,55,[illegible]
```



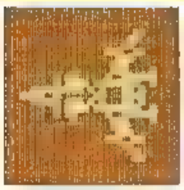
SPACE FIGHTER



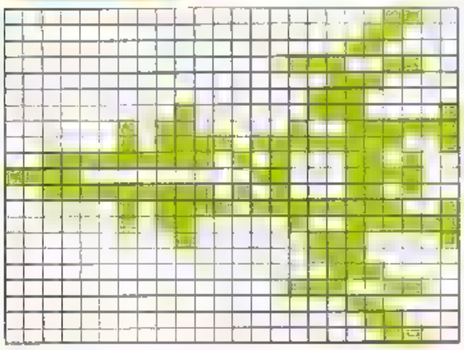
```
1,255,224,0,3,128,0
3,128,0,3,192,0,3
224,0,7,255,0,7,255
[illegible],3,240,48,28,8,127
[illegible],252,255,239,252,127,227
252,48,28,8,0,3,240
[illegible],255,0,7,255,[illegible]
[illegible],224,0,3,192,0,3
128,0,3,128,1,255,224
```



SPACE FIGHTER

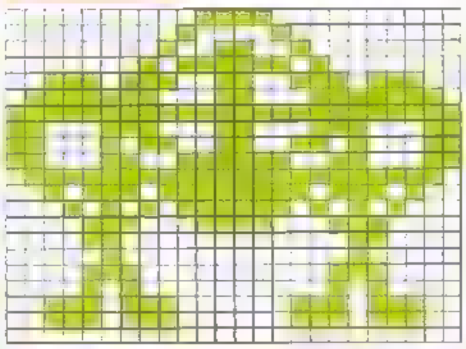


```
0,0,12,0,0,24,0
0,48,0,1,204,0,0
117,0,0,224,0,64,225
[illegible],191,[illegible],2,247,247,63
214,62,242,[illegible],60,62,214
52,2,247,249,[illegible],190,[illegible]
0,64,225,0,0,224,0
0,117,0,1,204,0,0
48,0,0,24,0,0,12
```

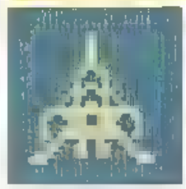


EXCURSION VEHICLE

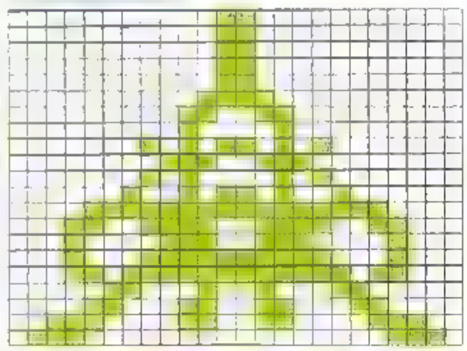
```
0,[illegible],0,0,[illegible],0,0
[illegible],[illegible],[illegible],126,179,59,[illegible]
[illegible],[illegible],24,[illegible],255,[illegible],255
199,[illegible],[illegible],19,[illegible],[illegible],[illegible]
126,59,127,255,254,46,255
[illegible],[illegible],[illegible],24,168,[illegible],60,112
12,0,[illegible],[illegible],[illegible],[illegible],[illegible]
0,112,[illegible],[illegible],60,59,129
220,59,129,[illegible],0,0,0
```



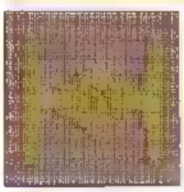
EXCURSION VEHICLE



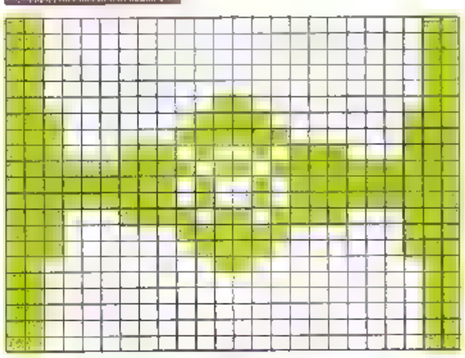
```
0,24,0,0,24,0,0
24,0,0,24,0,0,24
0,0,60,0,0,102,0
0,66,0,0,90,128,0
231,0,1,66,128,2,90
64,15,255,240,25,231,152
16,231,8,17,255,136,27
255,216,6,36,96,12,102
48,56,102,28,112,0,14
```



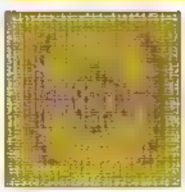
INTERGALACTIC CRUISER



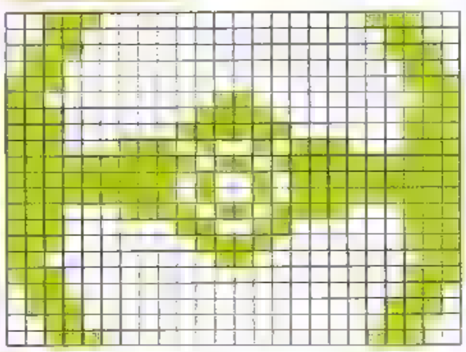
```
192,0,3,192,0,3,192
0,3,192,0,3,192,0
3,192,24,3,224,60,7
224,126,7,227,165,199,255
219,255,255,165,255,255,165
255,227,219,199,224,126,7
224,60,7,192,24,3,192
0,3,192,0,3,192,0
3,192,0,3,192,0,3
```



INTERGALACTIC CRUISER

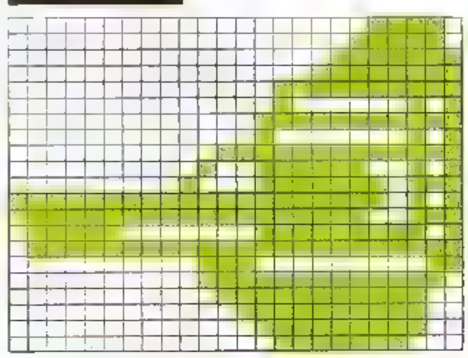


```
56,0,28,48,0,12,112
[illegible],14,96,0,6,224,0
[illegible],192,24,3,224,60,7
224,126,7,227,165,199,255
[illegible],255,255,165,255,255,165
255,227,219,199,224,126,7
224,56,7,192,24,3,224
0,[illegible],56,0,[illegible],112,0
14,48,0,12,56,0,28
```



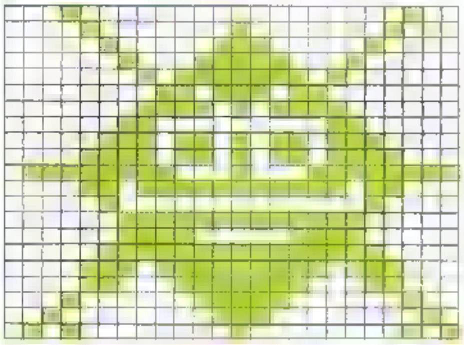
COMMAND SHIP

```
0,0,30,0,0,62,0
0,127,0,0,255,0,3
255,0,2,5,0,15,255
0,8,5,0,63,255,0
[illegible]9,[illegible],0,143,247,255,255
245,255,255,231,120,15,253
127,255,255,0,112,3,0
127,254,0,63,252,0,31
252,0,15,252,0,7,248
```



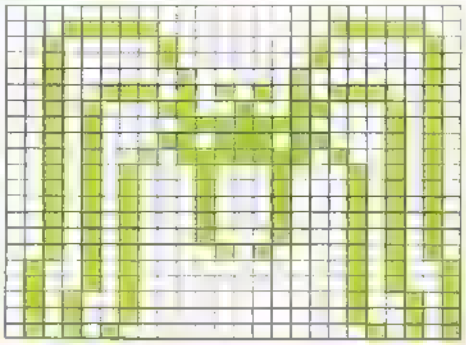
TRIBAL SPECTRE

16,3,4,8,8,8,4
28,16,2,62,12,1,[illegible]
64,0,221,126,1,235,192
3,0,36,7,207,[illegible],15
207,120,127,[illegible],255,11,255
[illegible],4,[illegible],16,3,255,224
3,193,224,7,255,240,14
255,189,12,127,24,16,62
4,32,28,2,90,9,5

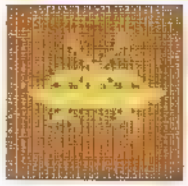


SPOOKY SPIDER

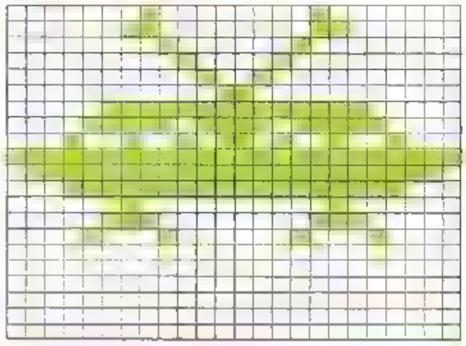
9,0,0,[illegible],0,124,32
128,130,32,128,130,32,65
2,[illegible],65,[illegible],40,251,132
40,127,[illegible],40,221,[illegible]
127,74,42,34,42,42,34
42,42,34,42,42,34,42
42,20,42,42,[illegible],42,74
0,41,74,[illegible],[illegible],82,0
37,82,0,37,84,0,21



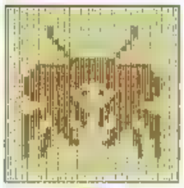
SPYING SAUCER



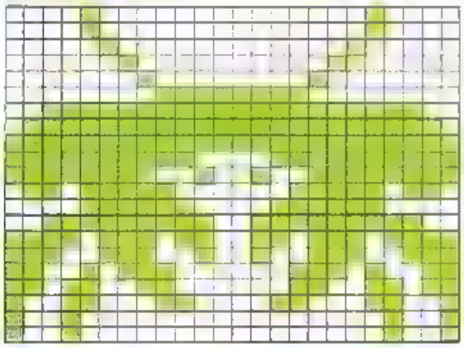
1,0,128,3,0,192,0
199,0,0,66,0,0,36
0,0,24,0,7,255,224
15,189,240,18,90,72,255
255,255,127,255,254,31,255
248,2,0,64,7,0,224
8,139,16,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0



SPACE CRAB

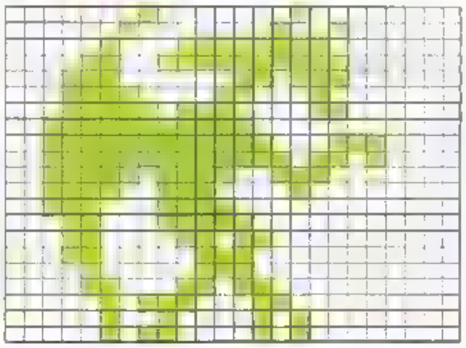


[illegible],[illegible],20,8,[illegible],16,4
0,32,2,0,64,1,0
128,[illegible],255,[illegible],61,[illegible],[illegible]
127,[illegible],224,[illegible],[illegible],255,255
1[illegible],251,255,36,255,[illegible],129
246,15,195,240,63,231,252
247,102,239,231,14,23[illegible],2[illegible]
1[illegible],[illegible],[illegible],[illegible],[illegible],[illegible],195
2[illegible],164,0,5,141,[illegible]

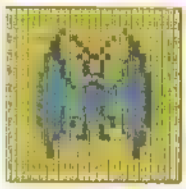


VAMPIRE

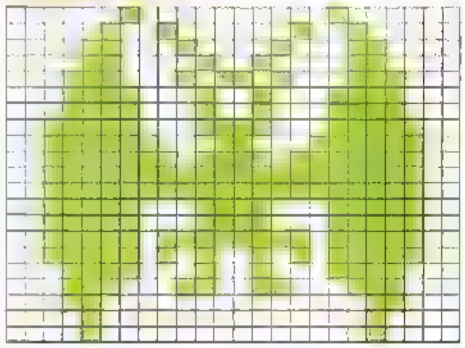
[illegible]
127,[illegible]8,12,2[illegible]5,1[illegible],[illegible]8,[illegible]
[illegible],[illegible]24,[illegible],[illegible],[illegible],192
[illegible],[illegible]40,0,63,252,112,63
254,2[illegible]8,62,2[illegible]1,128,60,255
[illegible],[illegible],24,[illegible],24,[illegible]4[illegible]
24,19[illegible],[illegible],24,[illegible]6,0,8
[illegible]0,0,12,[illegible]8,0,[illegible],198
[illegible]4,[illegible]33,0,5,133,[illegible]



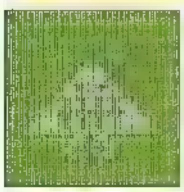
VAMPIRE



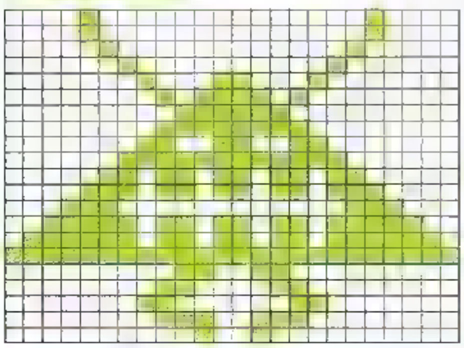
[illegible],[illegible],24,4,120,32,12
[illegible],48,12,36,45,28,[illegible]
16,3[illegible],36,120,30,36,120
62,[illegible],12[illegible],[illegible],24,252,6[illegible]
[illegible]
25[illegible],62,60,124,62,60,124
62,255,124,62,[illegible]65,124,30
24,[illegible]70,28,102,56,24,0
[illegible]4,[illegible],0,[illegible],[illegible],0,[illegible]



GHOUL

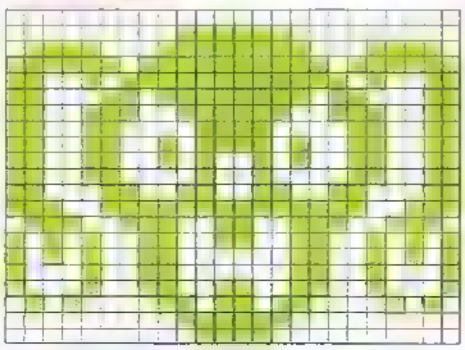


8,0,1[illegible],4,0,[illegible],4
0,32,2,0,64,1,24
[illegible]28,0,189,0,0,226,0
0,255,0,1,[illegible],128,3
255,1[illegible],[illegible],219,[illegible],14,219
112,28,0,55,62,219,124
[illegible]26,219,128,255,255,255,[illegible]
[illegible]32,0,0,195,[illegible],1,129
128,0,231,[illegible],[illegible],36,0

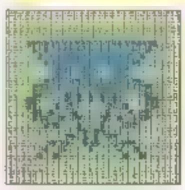


GHOUL

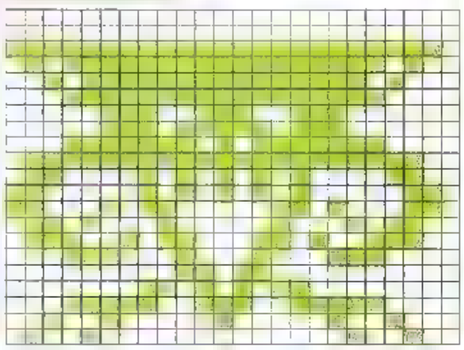
[illegible],126,0,120
25[illegible],30,253,255,[illegible],203,25[illegible]
[illegible]1,[illegible],255,95,2[illegible]6,126,[illegible]
[illegible],[illegible],[illegible],224,[illegible],[illegible],224
[illegible],[illegible],29[illegible],76,5[illegible],[illegible]98,[illegible]
79,[illegible],[illegible]5[illegible],[illegible]31,[illegible],[illegible],22[illegible]
[illegible]6[illegible],15[illegible],[illegible]49,[illegible]59,12[illegible],149,[illegible]
12[illegible],14[illegible],[illegible],[illegible]53,145,80,219
10,0,12[illegible],[illegible],[illegible],60,0



GHOUL

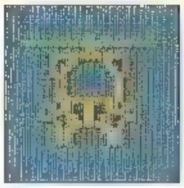


[illegible],[illegible],0,0,0,0,255
255,256,95,255,240,47,255
2[illegible],[illegible],255,240,7,57,192
7,5[illegible],[illegible]97,31,215,240,127
255,252,97,85,12,19[illegible],85
6,197,131,20,104,130,44
134,198,156,78,68,112,0
[illegible]8,0,[illegible],171,0,15,57
224,18,[illegible],144,36,0,72

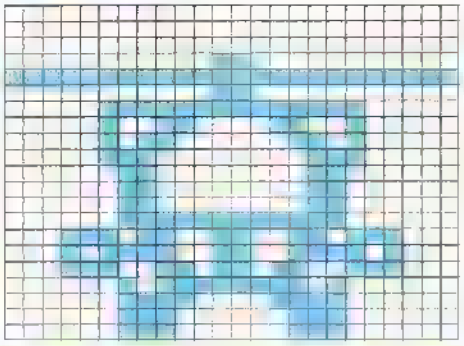


AIRCRAFT

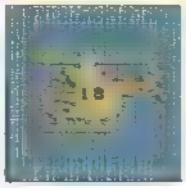
HELICOPTER



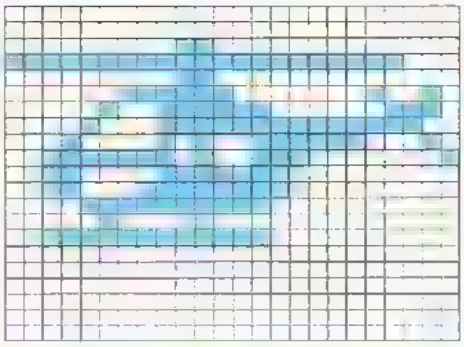
0,0,0,0,0,0,0
0,0,0,24,0,255,255
255,0,24,0,7,255,224
4,195,32,5,129,160,7
0,224,3,0,192,3,0
192,3,0,192,3,255,192
29,153,184,23,153,232,28
219,56,2,255,64,3,195
192,3,129,192,3,129,192



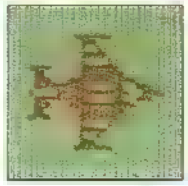
HELICOPTER



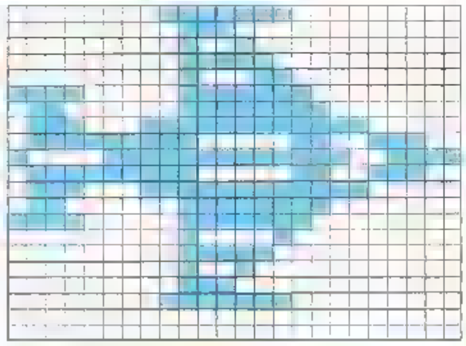
0,0,0,0,0,0,0
96,0,255,255,248,0,96
0,3,240,15,3,252,14
[illegible],255,252,16,167,236,63
167,1,48,255,1,48,255
[illegible],15,254,0,4,16,[illegible]
63,254,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0



HIGH-ALTITUDE JET

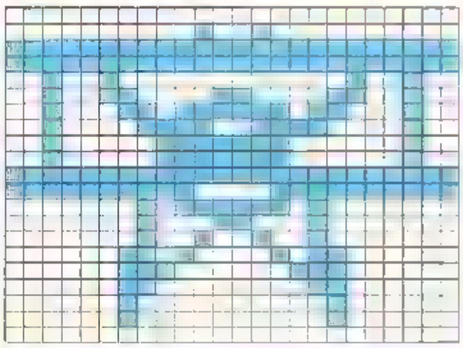


0,254,0,0,112,0,0
72,0,0,124,0,0,66
0,240,127,0,96,127,128
193,255,224,127,193,156,135
255,207,127,193,156,113,255
224,96,127,128,240,127,0
0,66,0,0,124,0,0
72,0,0,112,0,0,254
0,0,0,0,0,0,0

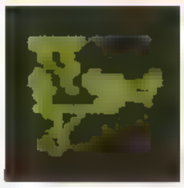


BIPLANE

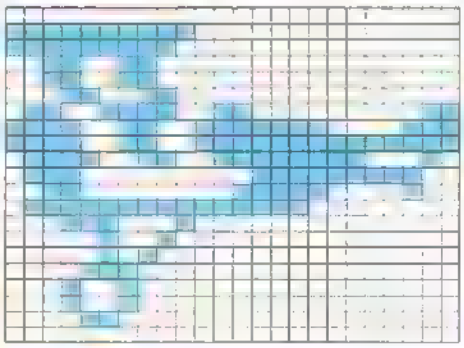
0,36,0,255,255,255,255
255,255,36,0,36,39,24
100,35,126,196,33,231,132
33,255,4,33,255,4,255
255,255,255,195,255,1,126
128,1,24,128,1,36,128
1,66,192,3,129,192,3
0,192,2,0,64,2,0
64,0,0,0,0,0,0



BIPLANE

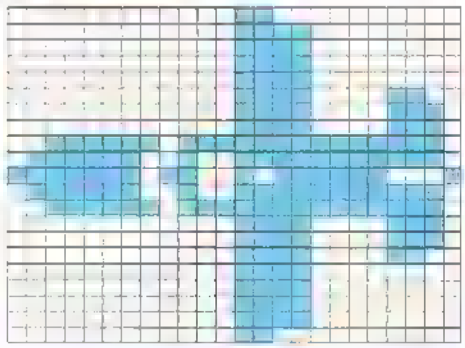


[illegible],0,0,255,[illegible],[illegible],255
128,0,51,0,0,51,0
[illegible],25,128,0,103,152,[illegible]
[illegible],63,135,240,63,255,[illegible]
15[illegible],226,112,7,252,48,15
132,127,255,0,28,64,0
[illegible],128,0,5,0,0,14
[illegible],0,16,0,0,[illegible],[illegible]
[illegible],12,[illegible],0,0,0

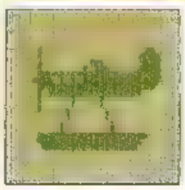


RECONNAISSANCE PLANE

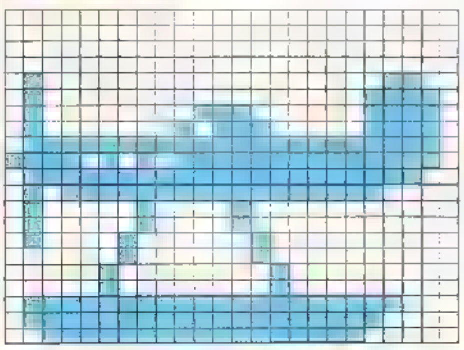
0,12,0,0,14,0,0
15,0,0,15,0,0,15
0,0,15,12,0,15,14
0,15,14,63,127,254,126
79,254,254,203,241,126,79
254,63,127,254,0,15,14
0,15,14,0,15,12,0
15,0,0,15,0,0,15
0,0,15,0,0,12,0



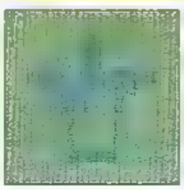
SEAPLANE



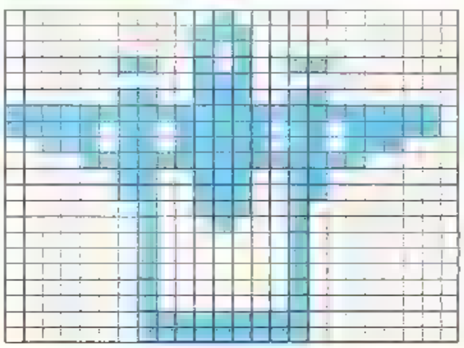
0,0,0,0,0,0,0
0,0,0,0,0,64,0
12,64,0,30,64,56,30
64,92,30,127,255,254,245
127,254,127,255,252,95,255
240,65,8,0,65,8,0
65,4,0,2,4,0,4
2,0,4,2,0,127,255
248,127,255,240,63,255,224



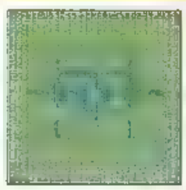
SEAPLANE



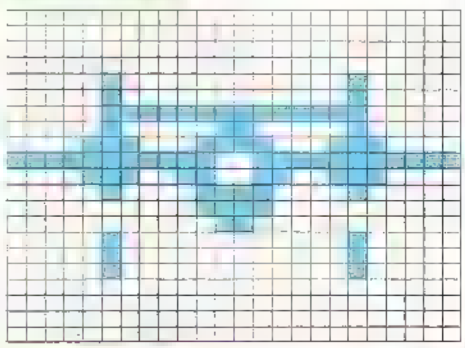
0,16,0,0,56,0,0
40,0,3,41,128,0,56
0,3,57,128,255,255,254
251,125,190,59,125,184,15
255,224,3,57,128,3,57
128,3,57,128,1,17,0
1,1,0,1,1,0,1
1,0,1,1,0,1,1
0,1,255,0,1,255,0



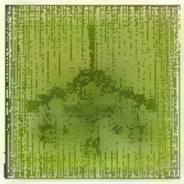
SEAPLANE



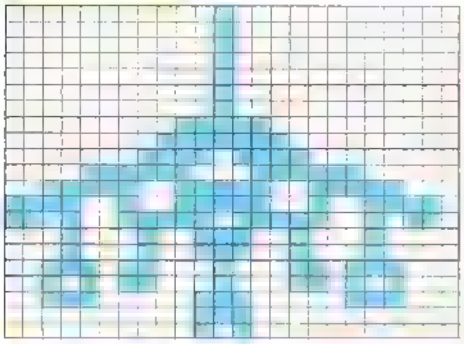
0,0,0,0,0,0,0
0,0,0,0,0,4,0
32,4,0,32,7,255,224
4,24,32,14,60,112,255
231,255,14,102,112,4,60
32,0,60,0,0,24,0
4,0,32,4,0,32,4
0,32,0,0,0,0,0
0,0,0,0,0,0,0



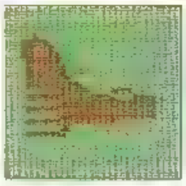
HIGH-ALTITUDE JET



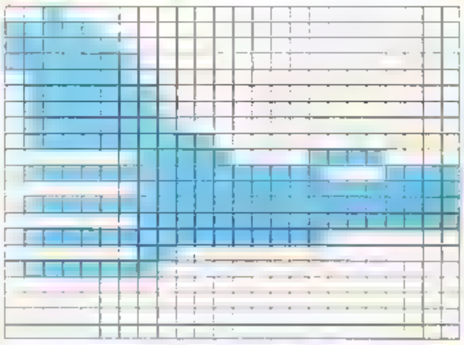
0,8,0,0,8,0,0
8,0,0,8,0,0,8
0,0,8,0,0,8,0
0,62,0,0,127,0,0
247,128,7,227,240,31,62
124,121,62,79,105,247,203
8,255,136,8,156,136,29
136,220,21,136,212,28,28
28,0,28,0,0,20,0



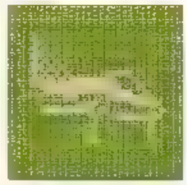
STUNT PLANE



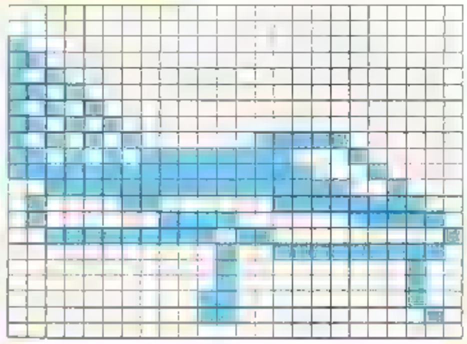
32,0,0,248,0,0,252
0,0,254,0,0,126,0
0,127,0,0,127,0,0
127,128,0,127,224,0,1
240,240,127,255,143,1,255
255,127,255,255,1,255,255
127,255,128,1,128,0,127
0,0,0,0,0,0,0
0,0,0,0,0,0,0



JET TRAINER

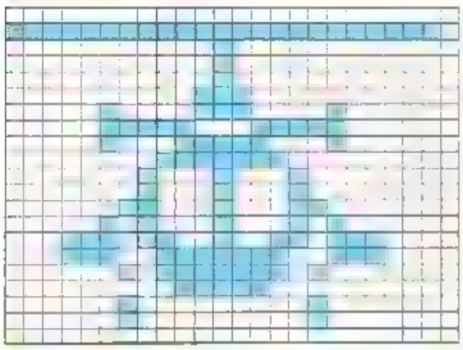


0,0,0,0,0,0,128
0,0,192,0,0,160,0
0,208,0,0,168,0,0
212,0,0,170,7,192,213
254,32,255,255,16,63,255
200,67,3,252,64,240,62
63,236,1,0,19,254,0
16,4,0,16,4,0,48
4,0,48,2,0,0,0

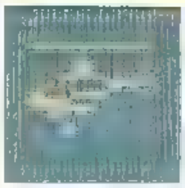


HELICOPTER

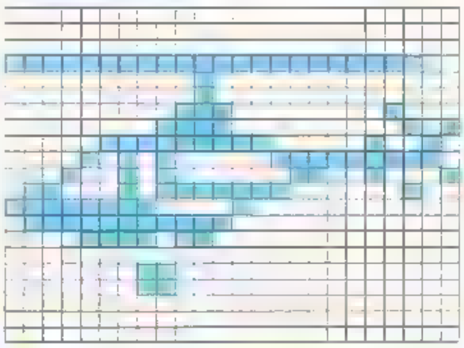
0,0,0,255,255,254,0
16,0,0,16,0,0,16
0,0,56,0,4,56,64
7,199,192,4,124,64,0
254,0,1,17,0,1,17
0,3,17,128,5,17,64
29,147,112,28,254,112,2
0,128,0,0,0,2,0
128,2,0,128,0,0,0



HELICOPTER

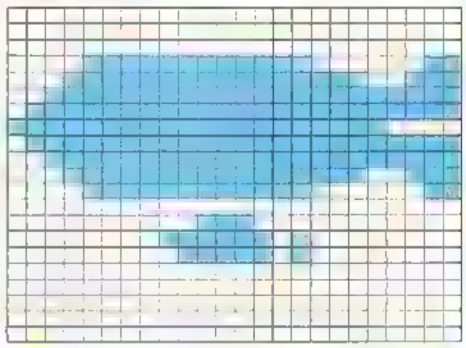


0,0,0,0,0,0,0
0,0,255,255,252,0,32
0,0,32,0,0,112,8
0,240,5,7,252,18,10
131,254,10,129,145,98,254
4,255,8,0,127,240,0
15,95,0,0,0,0,1
128,0,1,128,0,0,0
0,0,0,0,0,0,0

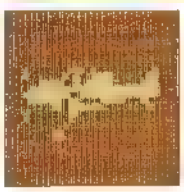


AIRSHIP

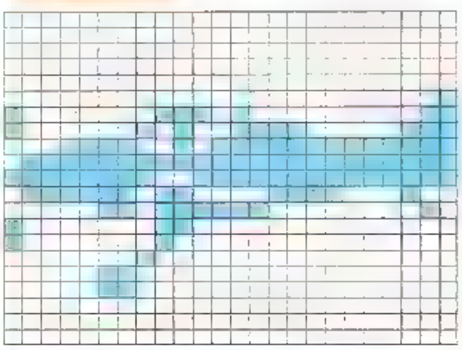
0,0,0,0,0,0,0
0,0,0,0,0,15,255
3,31,255,231,63,255,255
127,255,255,255,255,240,127
255,255,63,255,255,31,255
231,15,255,3,0,0,0
0,56,0,0,253,0,0
125,0,0,0,0,0,0
0,0,0,0,0,0,0



MONOPLANE

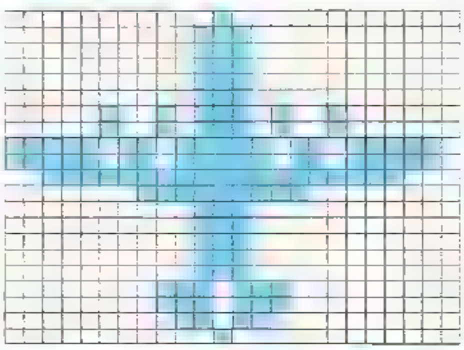


0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,128,232,1
129,79,3,32,95,7,127
255,255,255,63,255,126,195
195,6,252,2,128,192,0
128,128,0,1,0,0,6
0,0,6,0,0,0,0
0,0,0,0,0,0,0

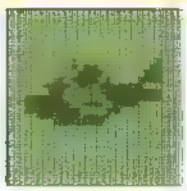


TRANSPORTER

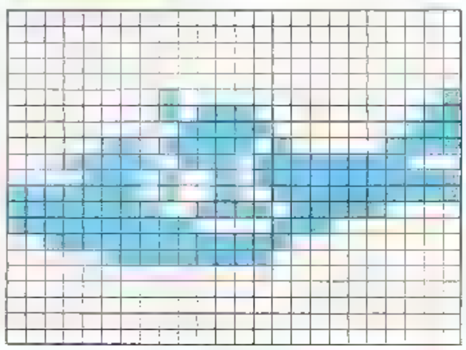
0,16,0,0,56,0,0
56,0,0,56,0,0,56
0,0,56,0,4,186,64
4,186,64,251,125,190,251
125,190,63,255,248,1,255
0,0,56,0,0,56,0
0,56,0,0,56,0,0
16,0,0,238,0,0,238
0,0,108,0,0,16,0



TRANSPORTER

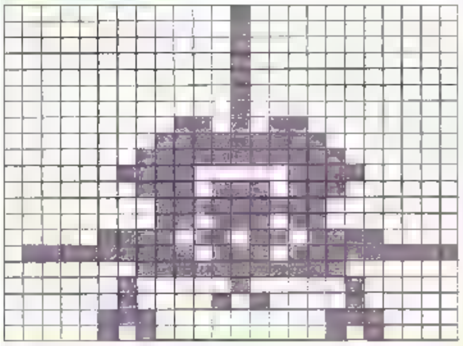


0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,128,1,0,184,3
0,174,7,5,250,15,31
119,248,14,151,255,254,147
255,255,61,240,255,195,128
63,254,0,7,252,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0

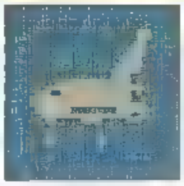


SHUTTLE

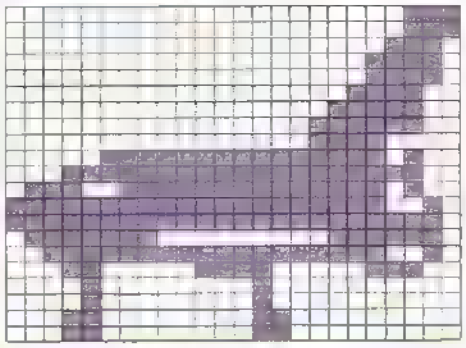
0,8,0,0,8,0,0
8,0,0,8,0,0,8
0,0,8,0,0,8,0
0,107,0,0,255,128,1
255,192,3,193,224,1,221
192,0,255,128,0,221,128
7,182,240,127,221,255,1
255,192,2,8,32,2,20
32,7,0,112,5,0,80



SHUTTLE

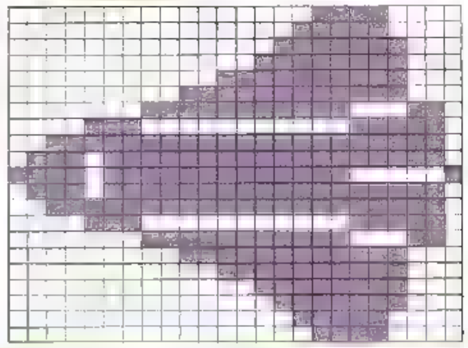


0,0,7,0,0,0,0
0,14,0,0,30,0,0
30,0,0,63,0,0,124
0,0,252,0,0,240,7
255,244,31,255,252,115,255
244,255,255,242,255,255,254
127,0,126,63,255,242,24
60,28,8,4,0,8,4
0,24,6,0,24,6,0

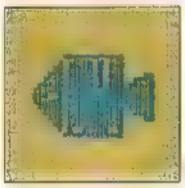


SHUTTLE

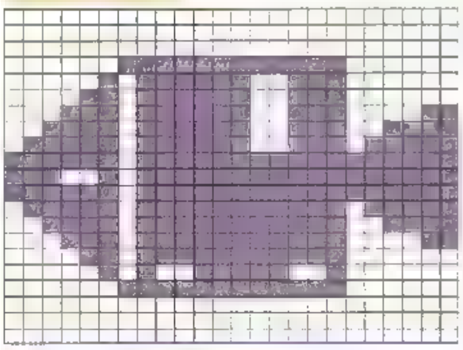
0,0,240,0,1,240,0
3,240,0,7,240,0,31
240,0,127,248,1,255,198
14,0,62,63,255,254,119
255,254,247,255,193,119,255
254,63,255,254,14,0,62
1,255,198,0,127,248,0
31,248,0,7,248,0,3
248,0,1,240,0,0,240



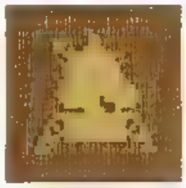
LUNAR MODULE



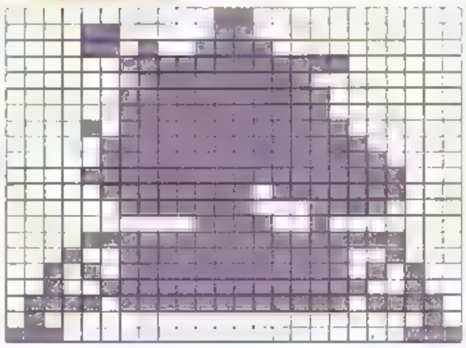
0,0,0,0,0,0,0
0,0,3,255,192,3,245
192,13,245,192,29,245,195
61,245,207,125,245,223,253
255,255,220,255,255,253,255
255,125,255,223,61,255,207
29,255,195,13,255,192,5
62,64,3,255,192,0,0
0,0,0,0,0,0,0



LUNAR LANDER

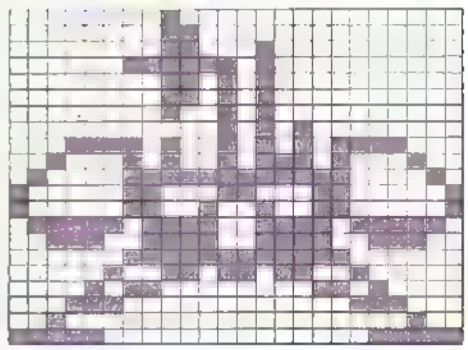


0,24,0,12,24,0,15
60,0,3,255,192,1,255
0,3,255,128,3,255,128
15,255,192,7,255,224,7
255,240,15,255,248,7,251
248,3,248,240,0,60,128
15,255,248,19,255,200,43
255,212,39,255,228,123,255
222,54,0,2,224,0,7



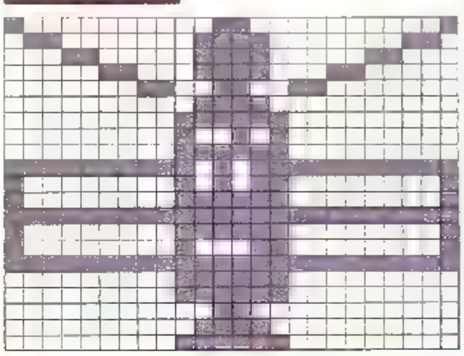
VIKING

0,128,0,0,192,0,0
196,0,3,244,0,0,212
0,0,212,0,0,148,0
0,21,0,30,149,120,34
151,68,67,255,194,131,24
193,131,36,193,125,231,190
57,255,156,3,255,192,4
219,32,11,0,208,28,0
56,32,0,4,248,0,31

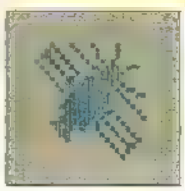


SKYLAB

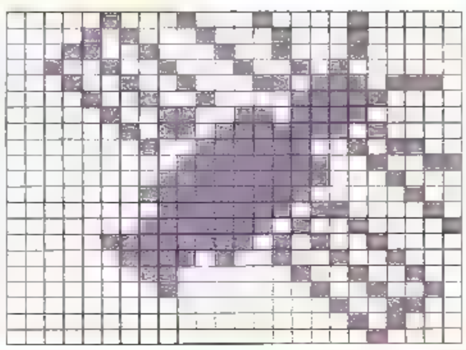
128,24,1,96,60,6,24
60,24,6,60,96,1,153
128,0,60,0,0,126,0
0,74,0,0,126,0,255
215,255,128,86,1,126,126
1,255,255,255,129,126,1
128,70,1,255,255,255,0
126,0,0,126,0,0,24
0,0,60,0,0,126,0



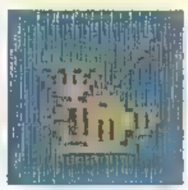
SKYLAB



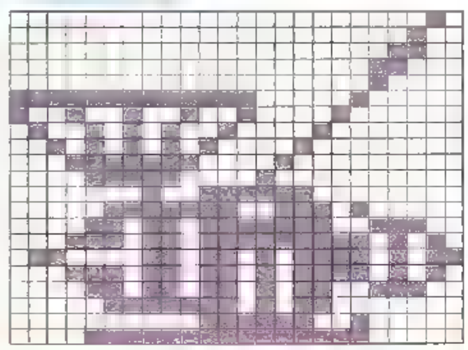
4,68,32,10,34,32,25
18,32,36,137,64,18,70
238,9,33,240,4,205,224
2,222,208,1,63,44,0
127,147,0,255,136,1,255
4,0,254,194,1,253,33
7,250,144,3,210,72,1
129,36,0,125,145,0,0
75,0,0,40,0,0,16



VENERA

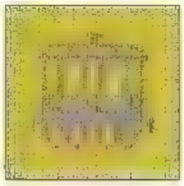


0,0,2,0,0,4,0
0,12,0,0,24,0,0
16,255,248,32,146,72,64
74,144,128,42,161,0,31
194,0,15,132,0,1,63
0,7,243,160,13,191,156
29,191,234,97,181,235,29
181,234,13,181,155,7,245
128,0,63,32,15,255,224

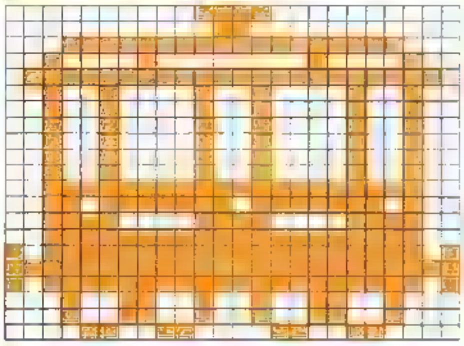


RAILWAY TRAINS

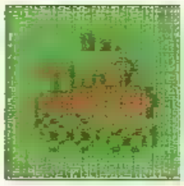
CARRIAGE



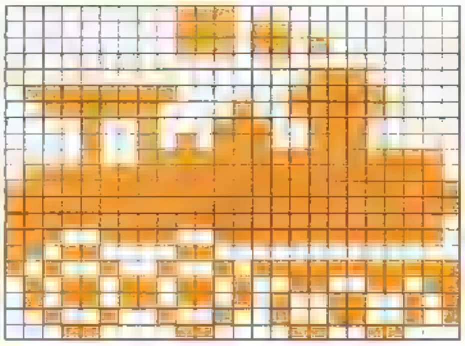
```
0,60,0,0,24,0,31
255,248,33,0,132,127,255
254,36,36,36,36,36,36
36,36,36,36,36,36,36
36,36,36,36,36,63,255
252,55,247,244,60,60,60
63,255,253,191,255,253,255
255,255,191,255,253,19,36
200,19,36,200,12,195,48
```



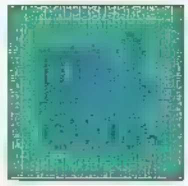
4-4-0 LOCO



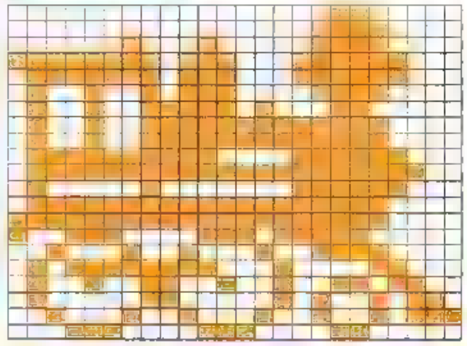
```
0,112,0,0,118,0,0
118,128,0,0,0,0,0
224,127,1,240,63,9,240
9,28,224,9,92,224,9
255,254,127,255,254,255,255
255,255,255,255,255,255,254
231,155,254,216,96,0,164
151,255,91,105,155,91,105
101,36,146,101,24,97,152
```



U.S. LOCO

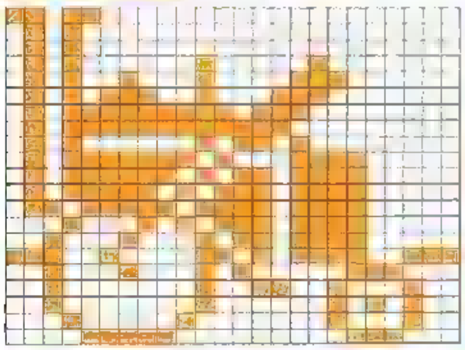


```
0,0,96,0,0,240,2
32,240,255,113,248,127,113
248,73,112,240,73,244,96
73,255,240,73,255,240,127
225,248,127,255,252,65,1
248,255,255,240,255,255,240
156,59,224,34,68,240,95
226,34,73,146,108,65,130
150,34,68,151,28,56,96
```



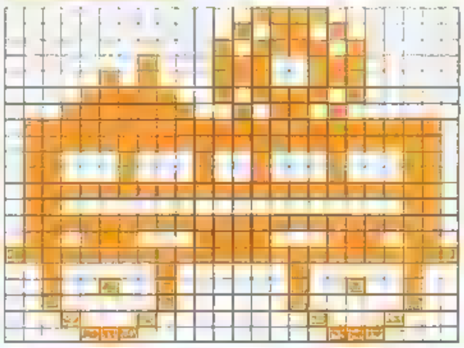
ROCKET

```
216,0,0,80,0,0,80
0,0,80,32,128,82,33
192,87,35,128,95,247,0
95,254,0,88,41,0,95
213,224,95,173,224,95,93
224,111,189,224,17,93,224
34,45,224,228,53,231,34
40,56,32,38,68,32,32
84,16,64,68,15,128,56
```

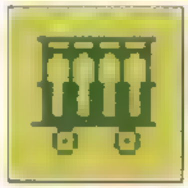


TENDER

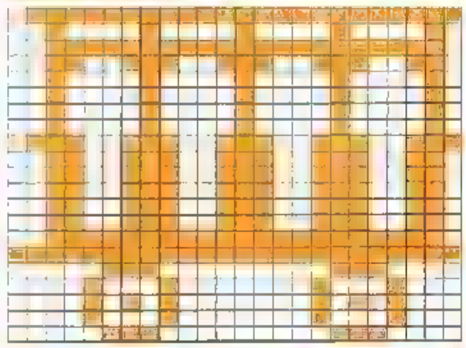
```
0,7,128,0,11,64,0
23,160,1,28,224,5,28
224,15,151,160,31,203,64
127,255,254,127,255,254,98
36,70,98,36,70,127,255
254,96,0,6,127,255,254
110,60,118,255,255,255,32
129,4,36,129,36,32,129
4,17,0,136,14,0,112
```



CARRIAGE

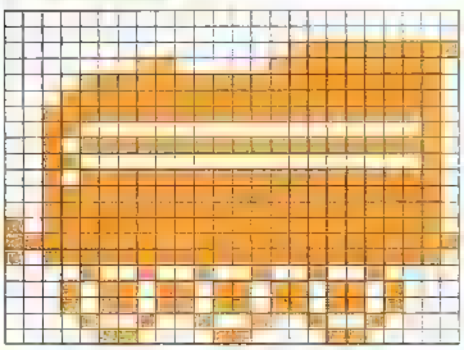


```
127,255,255,33,8,66,63
255,254,51,156,230,33,8
66,33,8,66,33,8,66
33,8,66,115,156,231,51
156,230,51,156,230,51,156
230,51,156,230,51,156,230
63,255,254,255,255,255,7
0,112,8,128,136,10,128
168,8,128,136,7,0,112
```



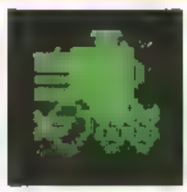
TENDER

```
0,0,0,0,0,0,0
1,255,1,131,254,15,255
254,31,255,254,63,255,254
48,0,2,47,255,254,48
0,2,47,255,254,63,255
254,63,255,254,191,255,254
255,255,255,191,255,254,9
36,144,22,219,104,22,219
104,9,36,144,6,24,96
```

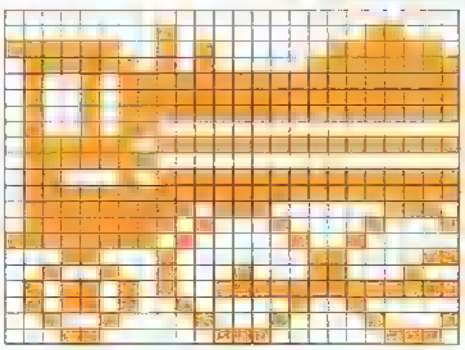


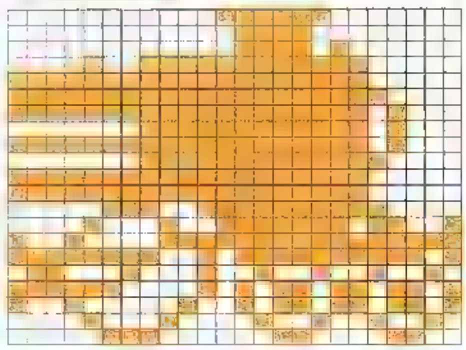
PACIFIC-TYPE LOCO

```
0,0,0,0,128,24,252
160,126,136,160,255,75,255
255,75,255,255,75,255,255
75,0,0,126,255,255,127
0,0,98,255,255,127,255
255,127,220,14,127,162,17
255,65,32,24,129,195,36
136,196,90,159,255,90,65
32,36,34,17,24,28,14
```



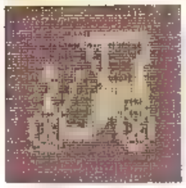
```
0,31,128,0,15,0,0
15,60,9,255,224,255,255
224,255,255,240,255,255,232
1,255,232,255,255,232,1
255,240,255,255,224,255,255
224,7,63,240,8,159,217
144,255,255,255,47,237,98
36,146,252,43,109,144,75
109,8,132,146,7,3,12
```



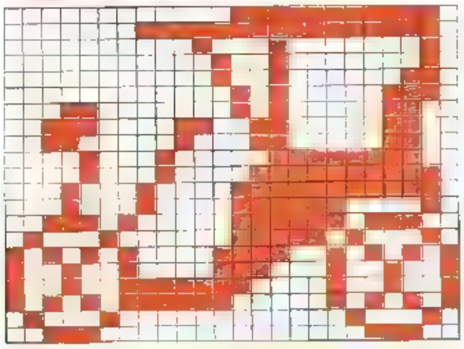


CARS, TRUCKS AND MOTORBIKES

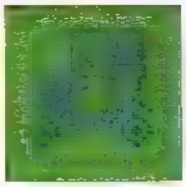
VETERAN



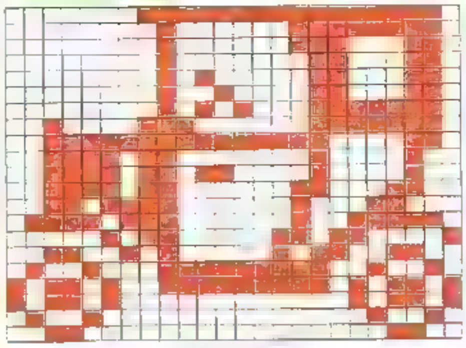
```
0,15,254,1,255,254,0
34,2,0,18,2,0,18
5,0,10,14,24,10,12
56,102,12,48,70,12,8
131,252,8,135,254,17,15
254,17,15,194,58,15,220
68,15,162,170,30,85,146
28,73,147,248,73,171,240
85,68,0,34,56,0,28
```



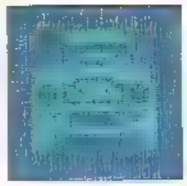
VETERAN



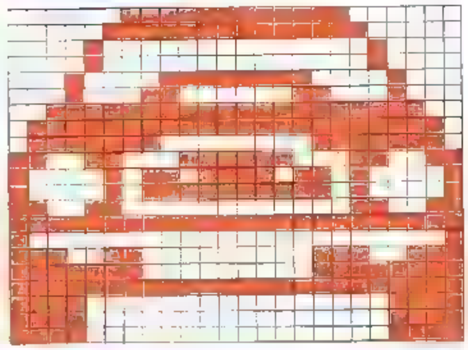
```
1,255,254,0,129,254,0
129,139,0,128,138,0,160
138,0,144,198,0,168,254
32,192,254,63,255,134,61
122,132,61,176,132,61,129
140,55,129,24,123,129,62
5,131,97,50,130,204,72
255,146,183,255,173,180,0
45,72,0,18,48,0,12
```



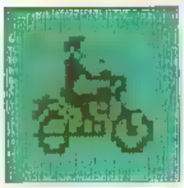
SALOON



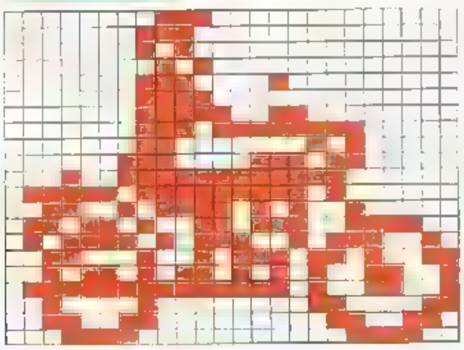
```
3,255,192,7,255,224,8
0,16,8,0,16,16,255
8,17,129,136,63,255,252
127,255,254,127,0,254,204
126,51,133,189,161,133,255
161,204,0,51,255,255,255
128,0,1,230,0,103,254
0,127,255,255,255,240,0
15,240,0,15,240,0,15
```



MOTORBIKE



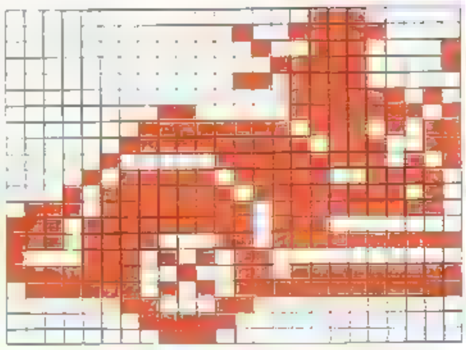
```
0,224,0,1,192,0,1
64,0,0,160,0,1,194
0,1,224,0,1,240,0
1,223,224,1,128,160,3
135,96,19,255,192,125,255
64,50,245,32,109,55,60
94,185,114,49,46,211,33
106,153,127,254,153,33,0
195,51,0,102,30,0,60
```

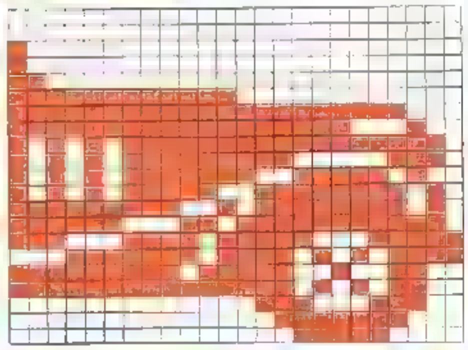


CLASSIC TOURER

```
0,0,224,0,0,240,0
4,225,0,3,225,0,13
224,0,1,250,0,191,253
1,254,235,3,255,115,4
31,245,11,237,159,23,247
250,219,251,254,239,251,125
239,27,255,152,172,0,254
77,255,126,175,255,3,24
0,1,240,0,0,224,0
```

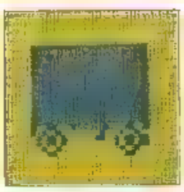
```
0,0,0,0,0,0,128
0,0,128,0,0,192,0
0,255,240,0,255,255,248
255,255,136,171,255,254,171
254,14,171,249,244,171,231
250,255,151,250,252,111,254
127,223,31,31,222,175,255
190,76,255,254,172,0,7
28,0,3,248,0,1,240
```

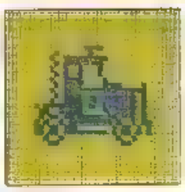




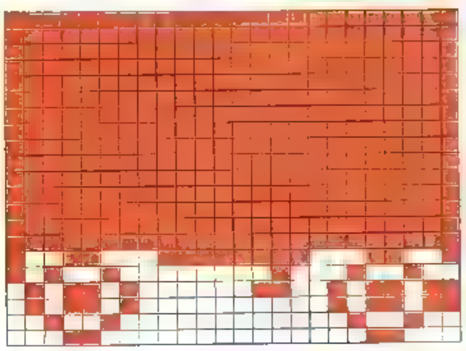
TRUCK

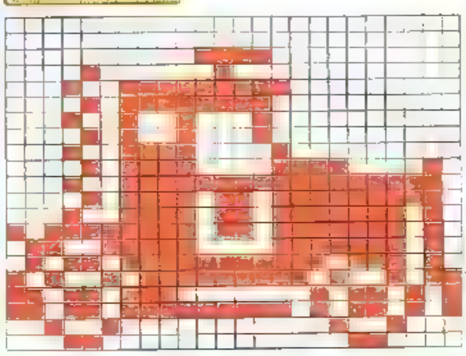
```
255,255,255,255,255,255,255
255,255,255,255,255,255,255
255,255,255,255,255,255,255
255,255,255,255,255,255,255
255,255,255,255,255,255,255
255,255,255,255,255,255,255
255,255,255,152,255,25,164
2,36,90,6,91,90,0
91,36,0,36,24,0,24
```





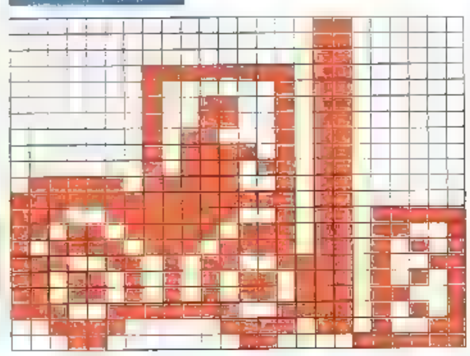
```
0,0,0,0,0,0,0
60,0,8,16,0,19,254
0,11,252,0,18,68,0
10,68,0,19,194,0,11
195,194,19,255,254,11,219
250,19,219,250,191,219,250
103,195,250,91,255,186,165
255,75,219,128,181,219,254
181,36,0,72,24,0,48
```





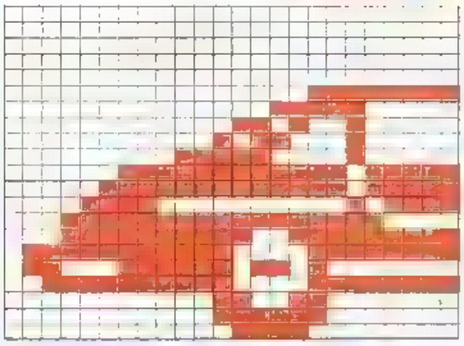
FORKLIFT

```
0,0,192,0,0,192,0
0,192,1,254,192,1,2
192,1,50,192,1,50,192
1,98,192,1,122,192,1
118,192,125,114,192,255,254
192,231,242,223,219,239,217
189,222,213,230,179,211,219
173,213,218,173,217,231,243
213,60,30,211,24,12,63
```

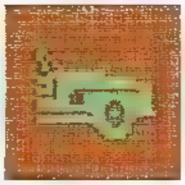


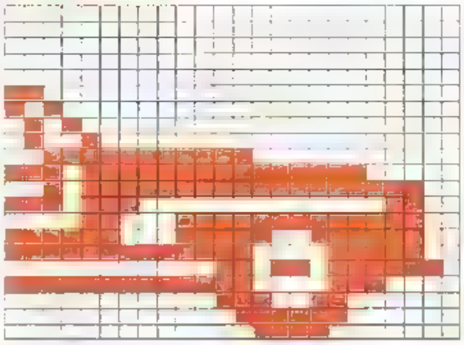
SPORTS SALOON

```
0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,255,0,3,224
0,13,32,0,22,32,0
124,32,3,255,223,7,255
223,15,0,47,31,255,240
31,249,255,127,240,255,99
246,128,63,240,255,0,25
128,0,31,128,0,15,0
```

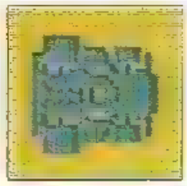


```
0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,192,0,0,160,0,0
80,0,0,72,0,0,63
248,0,239,255,224,47,255
252,239,0,12,12,127,252
252,124,252,255,248,124,0
77,126,255,248,96,0,28
192,0,15,192,0,7,128
```

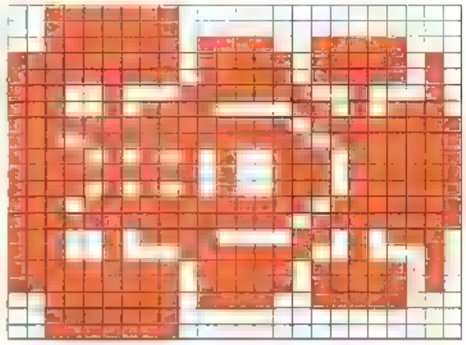




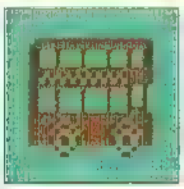
FORMULA 1



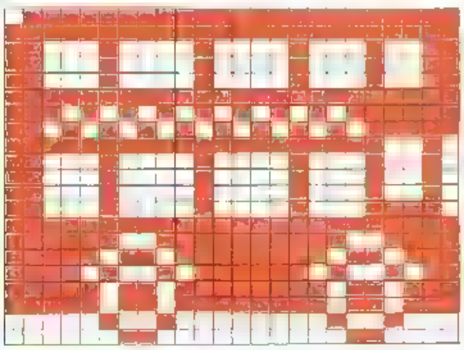
```
63,128,0,63,128,0,63
188,248,255,190,250,255,190
250,228,127,34,229,225,174
255,254,255,239,255,127,245
83,191,255,211,191,245,83
191,239,255,127,255,254,255
229,225,174,228,127,34,255
190,250,255,190,250,63,188
248,63,128,0,63,128,0
```



LONDON BUS

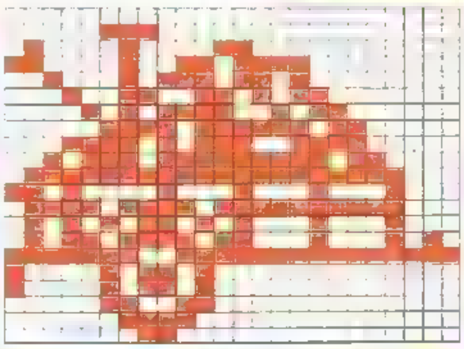


```
127,255,255,255,255,255,196
33,19,196,33,19,196,33
19,255,255,255,234,170,191
213,85,95,255,255,242,196
33,18,196,33,18,196,33
18,196,33,30,255,255,255
252,255,207,251,127,183,244
191,75,251,127,183,251,127
183,4,128,72,3,0,48
```

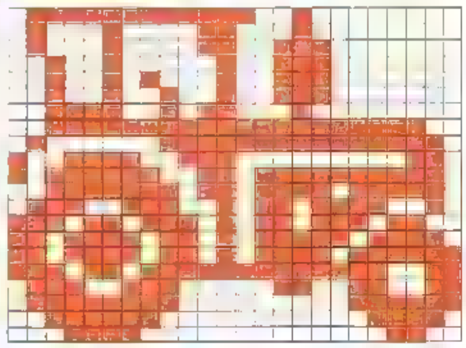


TRACTOR

```
0,0,0,7,0,0,66
24,0,194,111,0,34,237
0,19,53,128,10,242,192
7,191,96,30,185,224,46
255,176,127,255,248,209,136
136,123,223,248,05,168,136
91,216,138,254,127,255,133
160,0,133,160,0,6,96
0,3,192,0,1,128,0
```

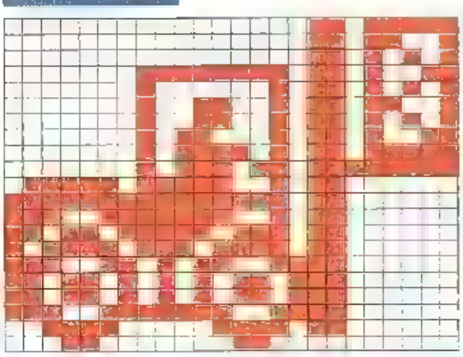


```
127,249,0,68,49,0,68
51,128,36,19,128,37,19
128,36,147,128,60,177,32
63,255,252,64,255,254,158
112,8,191,55,254,63,183
190,115,150,242,237,214,236
222,215,222,222,215,191,237
255,243,115,131,51,127,128
63,63,0,30,30,0,12
```

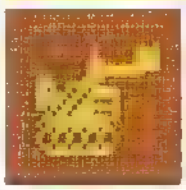


FORKLIFT

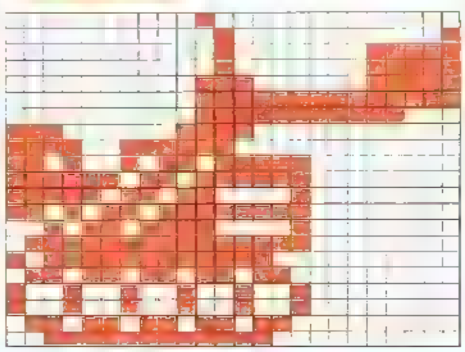
```
0,0,223,0,0,217,0
0,213,1,254,211,1,2
213,1,50,217,1,50,213
1,98,211,1,122,223,1
118,255,125,114,192,255,254
192,231,242,192,219,238,192
189,222,192,230,179,192,218
173,192,218,173,192,231,243
192,60,30,192,24,12,0
```



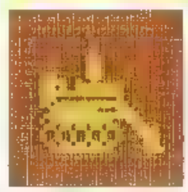
BULLDOZER



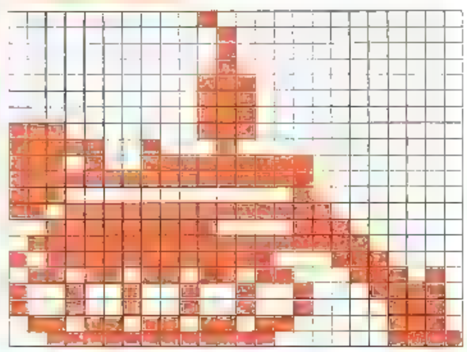
```
0,32,0,0,16,1,0
16,31,0,16,31,0,56
31,0,63,255,0,63,248
224,120,0,243,240,0,210
223,0,253,191,0,251,97
0,214,255,0,45,225,0
127,255,0,191,252,0,109
182,0,146,73,0,146,73
0,109,182,0,63,252,0
```



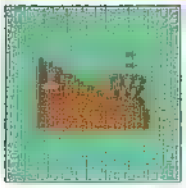
BULLDOZER



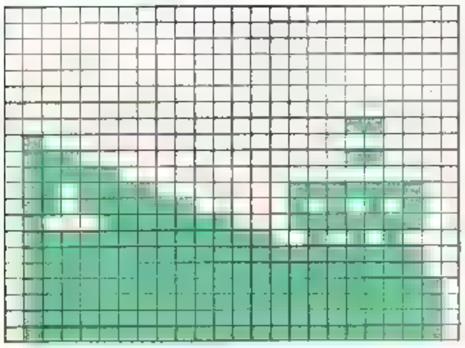
```
0,64,0,0,32,0,0
32,0,0,32,0,0,112
0,0,112,0,0,112,0
224,112,0,249,32,0,233
255,0,255,255,0,224,1
0,223,255,128,63,225,192
127,255,224,191,252,112,109
182,58,146,73,30,146,73
14,109,182,14,63,252,15
```



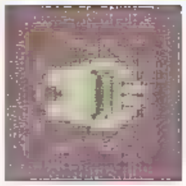
FREIGHTER



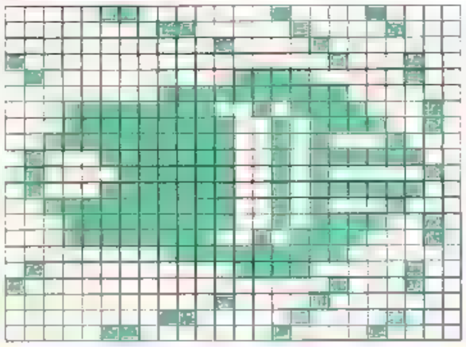
```
0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,48,64,0,0,112
0,48,124,0,0,111,1
252,111,193,84,71,241,252
127,252,168,127,255,252,127
255,252,127,255,254,127,255
254,127,255,254,127,255,254
```



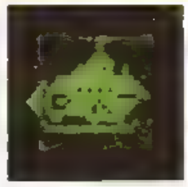
SPEEDBOAT (FROM ABOVE)



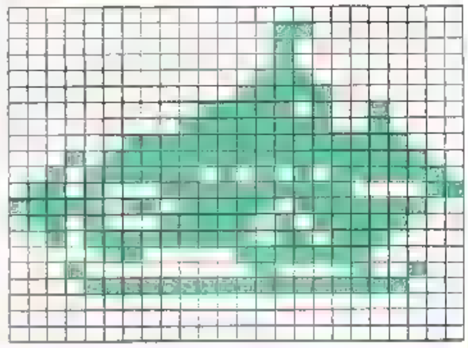
```
5,2,16,0,193,8,0
16,128,128,0,68,64,15
4,3,255,192,31,229,226
31,242,242,63,242,136,71
242,252,67,242,130,71,242
252,63,242,136,31,242,242
31,229,226,0,255,192,64
15,4,128,3,68,0,16
128,0,193,8,6,2,16
```



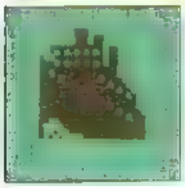
SUBMERSIBLE



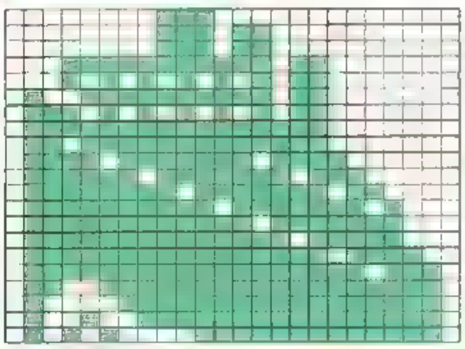
```
0,0,0,0,3,0,0
7,0,0,2,0,0,7
0,0,15,128,0,62,144
0,127,144,3,255,248,23
255,252,47,213,126,104,255
195,251,125,252,104,250,248
47,251,112,6,15,224,16
4,36,15,255,248,0,0
0,0,0,0,0,0,0
```



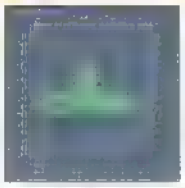
LINER



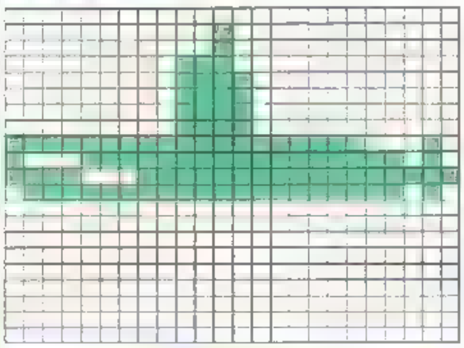
```
0,224,0,0,236,0,0
236,0,31,253,128,21,85
128,79,253,128,117,85,128
127,255,128,111,255,128,123
251,192,126,254,224,127,191
176,127,239,232,127,251,248
127,254,248,127,255,188,127
255,238,103,255,254,67,255
254,73,255,254,84,63,254
```



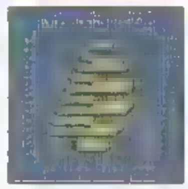
SUBMARINE



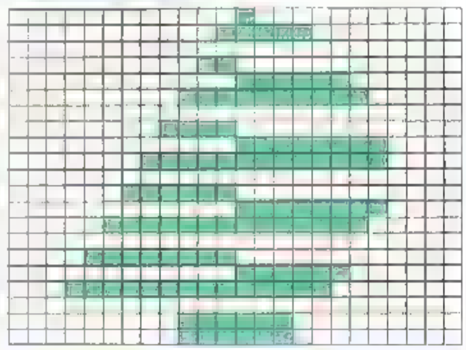
```
0,0,0,0,16,0,0
16,0,0,112,0,0,120
0,0,120,0,0,120,0
0,120,0,255,255,226,143
255,250,241,255,255,127,255
250,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0
```



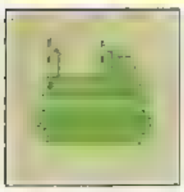
YACHT



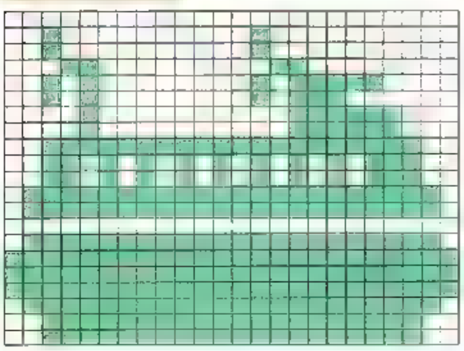
```
0,8,0,0,31,0,0
0,0,0,48,0,0,15
192,0,127,224,0,0,0
0,240,0,0,15,240,1
255,240,0,0,0,1,240
0,0,15,240,7,255,224
0,0,0,15,240,0,0
15,192,31,255,128,0,0
0,0,126,0,0,126,0
```



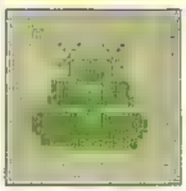
HOVERCRAFT



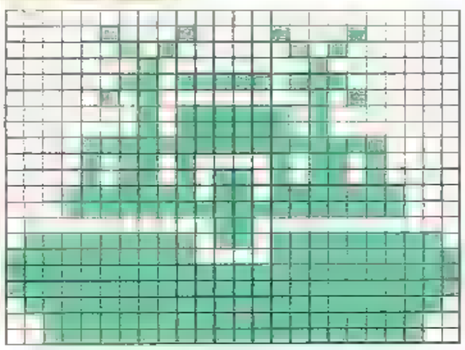
```
0,0,0,32,4,0,32
4,0,24,1,0,40,5
240,40,5,192,8,1,248
8,1,248,63,255,252,53
85,92,53,85,92,127,255
254,127,255,254,0,0,0
127,255,254,255,255,255,255
255,255,255,255,255,127,255
254,63,255,252,63,255,252
```



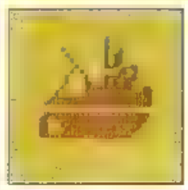
HOVERCRAFT



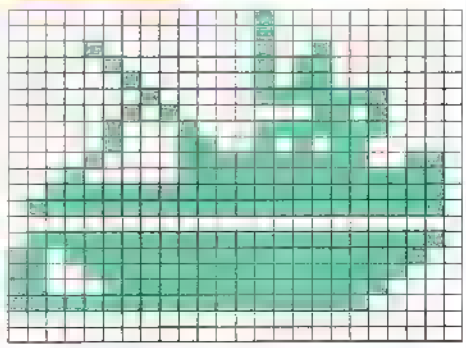
```
0,0,0,4,66,32,2
129,64,1,0,128,3,189
192,5,0,160,1,126,128
1,126,128,15,255,240,11
56,208,11,90,208,31,219
248,31,255,248,0,24,0
127,219,254,255,195,255,255
255,255,255,255,255,127,255
254,63,255,252,63,255,252
```



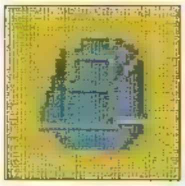
TUGBOAT



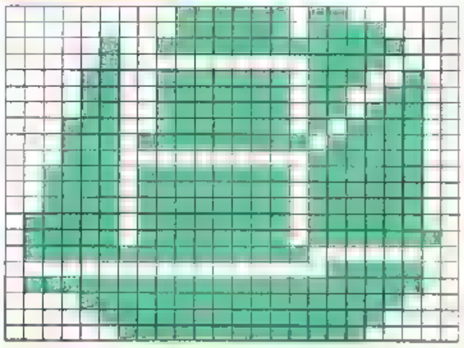
```
0,4,0,0,4,0,8
4,128,4,5,128,2,5
128,1,7,240,2,128,208
4,71,240,4,101,96,8
127,230,16,127,254,63,255
254,127,255,254,0,0,0
127,255,254,223,255,252,207
255,248,199,255,240,255,255
224,0,0,0,0,0,0
```



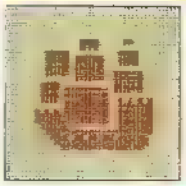
TALL SHIP



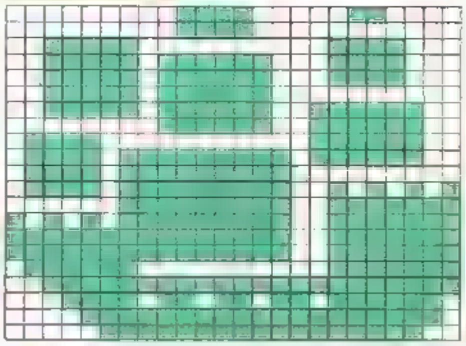
```
0,126,192,0,126,240,4
255,248,4,0,240,12,254
228,12,254,204,12,254,220
28,254,188,29,255,124,28
0,252,61,254,252,61,254
252,61,254,252,125,254,252
125,254,254,127,255,0,0
0,124,127,255,252,15,255
248,7,255,240,3,255,224
```



MAN O' WAR

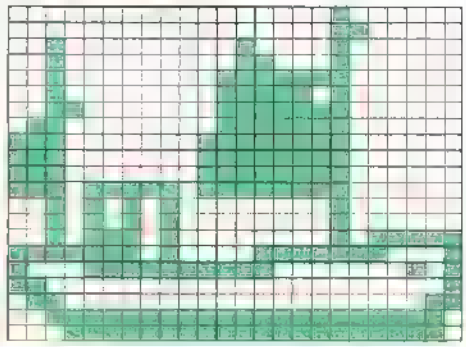


```
0,120,48,0,120,0,62
0,120,62,252,120,62,252
127,62,252,0,62,252,252
0,252,252,120,0,252,123
254,252,123,254,0,123,254
65,3,254,127,171,254,127
255,254,127,255,254,127,126
0,127,63,255,255,30,219
126,15,255,254,7,255,252
```

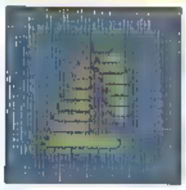


FISHING SMACK

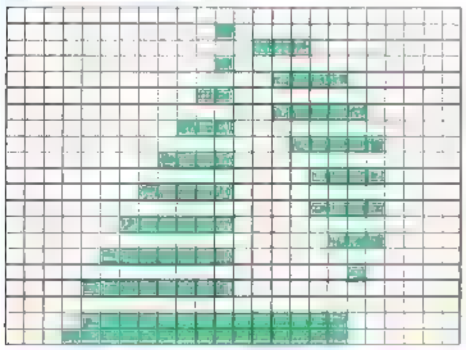
```
0,0,64,0,0,96,32
8,64,32,12,64,32,31
192,48,31,64,96,31,192
224,31,192,224,63,192,224
63,192,224,63,192,47,191
192,42,128,64,42,128,64
46,128,95,254,135,241,143
252,5,192,0,1,96,0
3,63,255,254,31,255,254
```



YACHT

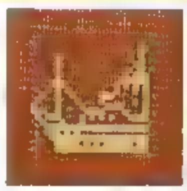


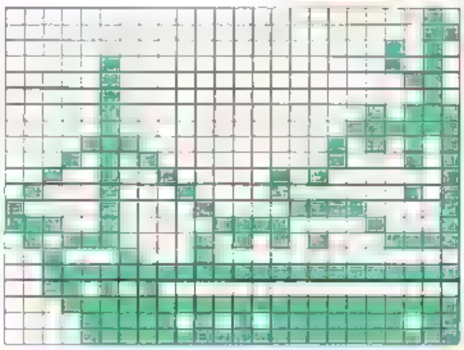
```
0,0,0,0,16,0,0
7,0,0,16,0,0,3
192,0,48,0,0,3,224
0,112,0,0,1,240,0
240,0,0,0,240,[illegible],240
0,0,0,240,3,240,0
0,0,112,7,240,0,0
0,32,15,240,0,0,0
0,15,255,192,31,255,192
```



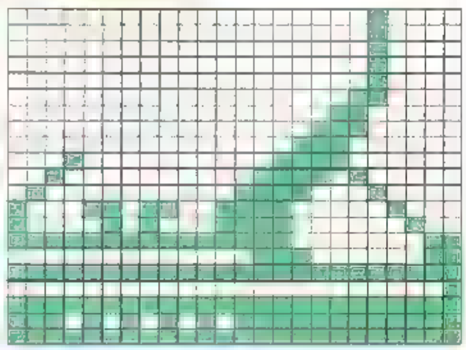
STERN TRAWLER

```
0,0,2,0,0,3,0
0,16,4,0,16,4,0
6,4,0,2,4,0,23
4,3,63,14,0,65,21
0,59,36,131,129,68,66
5,132,35,225,228,33,17
92,47,143,34,32,0,11
255,255,29,64,0,15,255
255,7,213,251,7,255,255
```



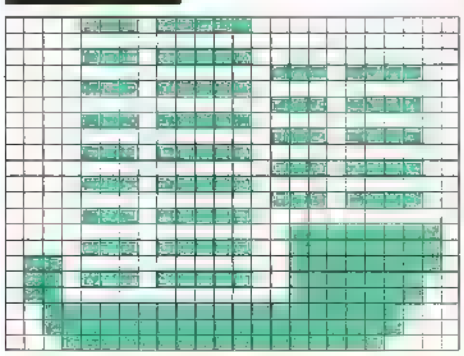


```
0,0,16,0,0,16,0
0,16,0,0,16,0,0
16,0,0,48,0,0,96
0,0,224,0,1,192,16
3,192,32,7,160,64,15
16,141,158,8,133,30,4
255,254,3,0,7,3,255
255,255,0,0,1,255,255
255,234,175,255,255,255,255
```



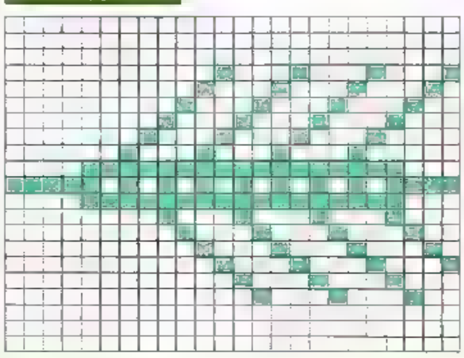
JUNK

```
14,240,0,0,0,0,14
248,0,0,3,188,14,248
0,0,3,188,14,248,0
0,3,188,14,248,0,0
3,188,14,248,0,0,3
188,14,248,0,0,1,254
14,249,254,96,1,254,110
249,252,96,1,252,63,255
248,63,255,248,31,255,240
```



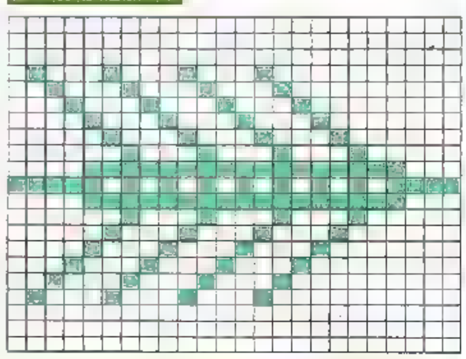
ROWING EIGHT

```
0,0,0,0,0,0,0
0,0,0,17,17,0,34
34,0,68,68,0,136,136
1,17,15,2,34,32,15
255,248,250,170,175,15,255
248,0,136,136,0,68,68
0,34,34,0,17,17,0
8,136,0,4,68,0,0
0,0,0,0,0,0,0
```



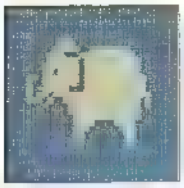
ROWING EIGHT

```
0,0,0,0,0,0,0
0,0,68,68,0,34,34
0,17,17,0,8,136,128
4,68,64,2,34,32,15
255,248,250,170,175,15,255
248,2,34,32,4,68,64
8,136,128,17,17,0,34
34,0,68,68,0,0,0
0,0,0,0,0,0,0
```

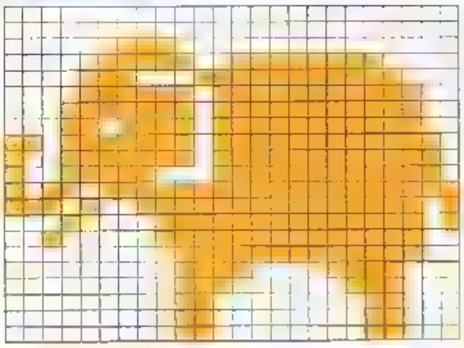


ANIMALS

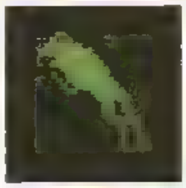
ELEPHANT



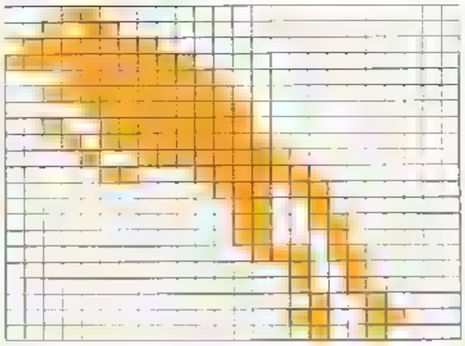
0,0,0,0,0,0,3
136,0,7,207,243,14,63
248,15,223,252,31,223,252
27,223,252,223,223,254,159
223,254,159,[illegible],254,187,255
255,239,255,[illegible],[illegible],254,[illegible]
32,255,252,0,127,252,[illegible]
170,120,0,112,56,0,56
24,0,96,24,0,96,24



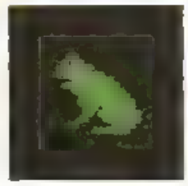
FROG



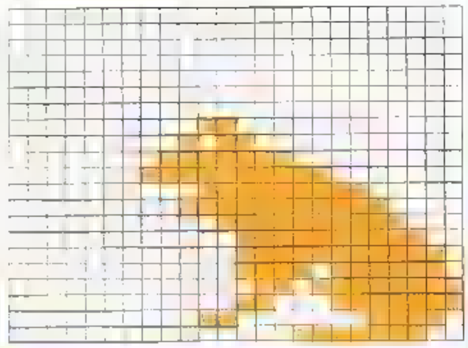
[illegible]



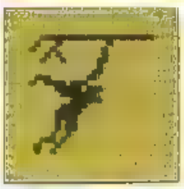
FROG



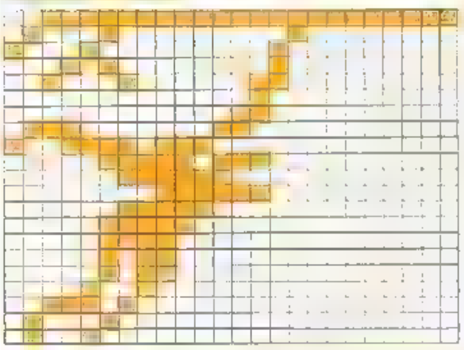
[illegible]



GIBBON

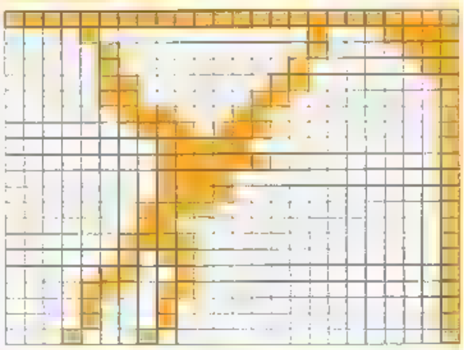


[illegible]

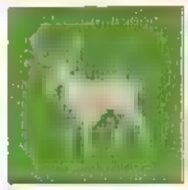


GIBBON

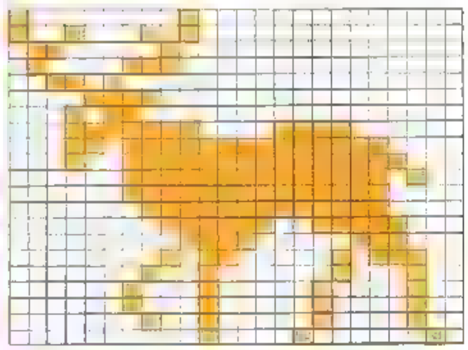
[illegible]



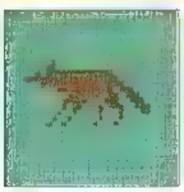
MOOSE



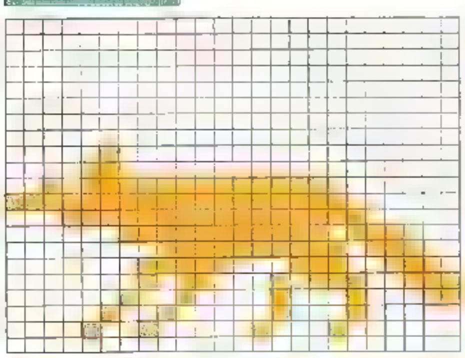
[illegible]



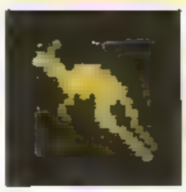
FOX



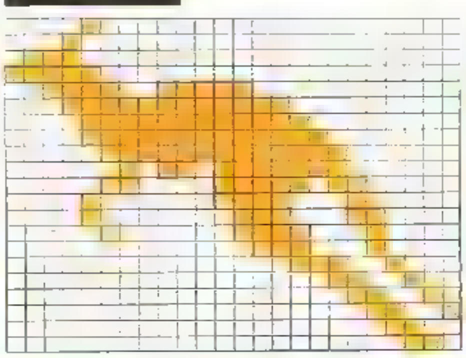
[illegible]



KANGAROO

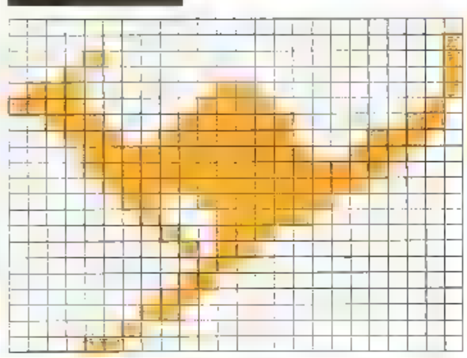


[illegible]

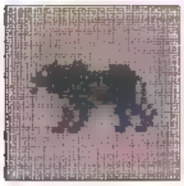


KANGAROO

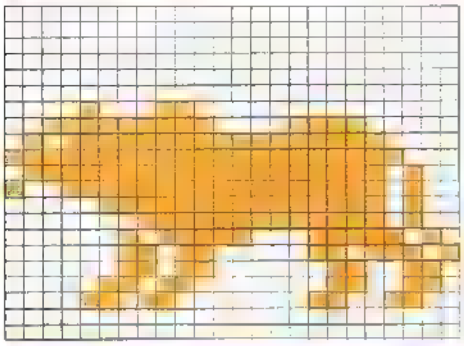
[illegible]



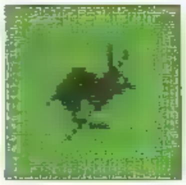
TIGER



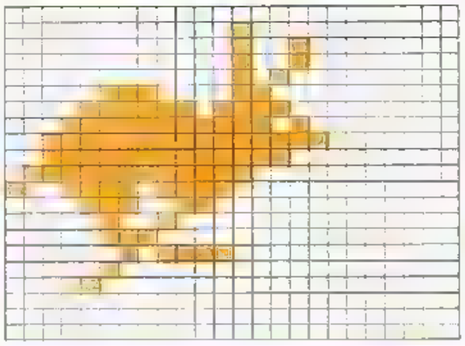
0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,0,0,8,192,0
[illegible],224,[illegible],255
255,248,127,[illegible],244,159,[illegible]
244,7,255,241,1,255,244
2,248,250,3,96,221,3
76,[illegible],6,192,[illegible],128
204,0,0,0,0,0,0



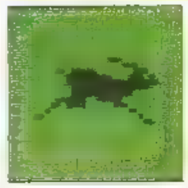
RABBIT



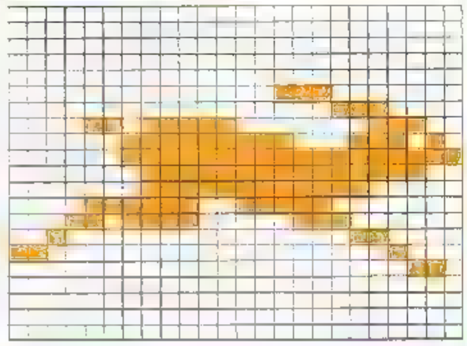
[illegible]
[illegible]
[illegible],15,222,[illegible]
[illegible],253,[illegible],63,255,125,63
254,0,127,248,[illegible],158,240
[illegible],14,76,[illegible],4,125,[illegible]
[illegible],2,240,[illegible]
0,0,8,0,0,0,0
0,0,0,0,0,0,0



RABBIT

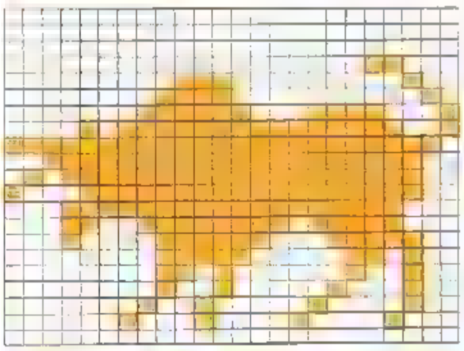


0,0,0,0,0,0,0
0,0,0,0,0,0,0
0,0,3,128,0,0,112
13,240,29,3,254,58,3
255,255,1,255,252,1,255
224,15,207,128,27,193,192
32,0,48,192,0,8,0
0,8,0,0,0,0,0
0,0,0,0,0,0,0



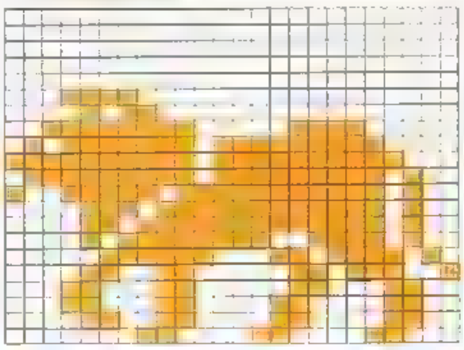
BUFFALO

[illegible]
[illegible]
4,0,[illegible],2,1,236,241
[illegible],255,[illegible],255,[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]

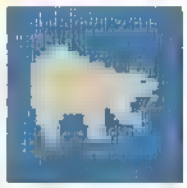


LION

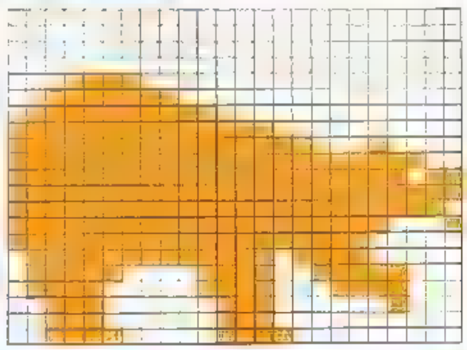
0,0,0,0,0,0,0
0,0,0,0,0,0,0
[illegible]
[illegible],191,224,[illegible],240,255
[illegible],248,127,[illegible]
[illegible],255,[illegible]
[illegible]
[illegible]
[illegible],12,[illegible],12,[illegible]



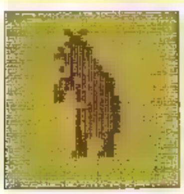
POLAR BEAR



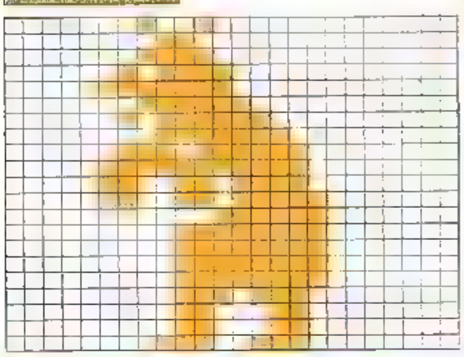
[illegible]
[illegible]
[illegible],128,[illegible]
[illegible],16,255
[illegible],255,[illegible]
[illegible],255,255,254,[illegible]
[illegible]
[illegible]
[illegible]



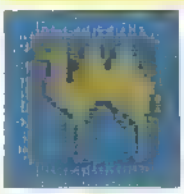
BROWN BEAR



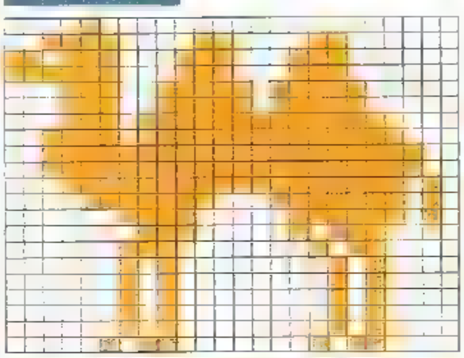
1,32,[illegible]
[illegible]
[illegible]
[illegible],254,[illegible]
[illegible]
128,[illegible],128,[illegible],128
[illegible],127,128,0,[illegible]
[illegible],128,[illegible],19,[illegible]
0,0,[illegible],231,[illegible]



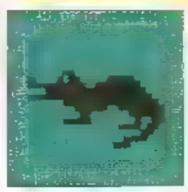
CAMEL



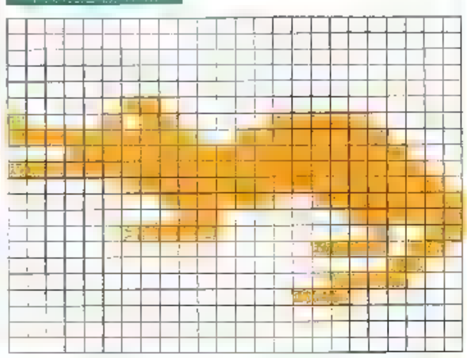
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
[illegible],128
[illegible],176



CROCODILE

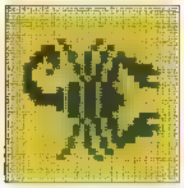


[illegible]
[illegible]
[illegible],133,131,224
[illegible]
[illegible]
[illegible],96,63,[illegible],224,19
[illegible]
[illegible]
0,0,0,0,0,0,0

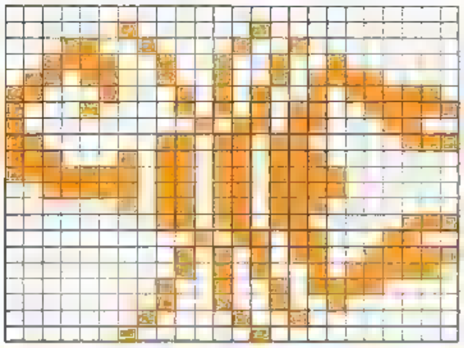


ANIMALS

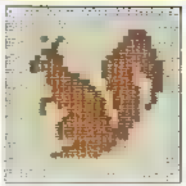
SCORPION



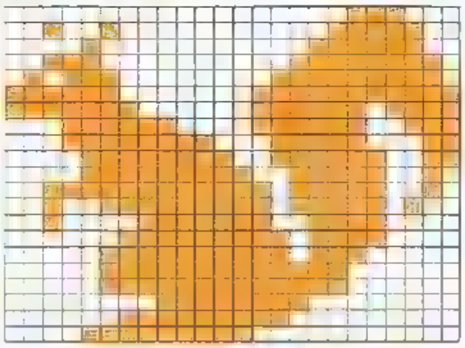
```
0,0,0,2,4,0,25
9,0,60,146,64,108,146
240,196,84,190,136,85,159
128,45,140,192,219,7,226
219,128,126,219,192,62,219
192,2,219,128,0,219,7
0,45,140,0,85,159,0
84,190,0,146,240,0,146
64,1,9,0,2,4,0
```



SQUIRREL

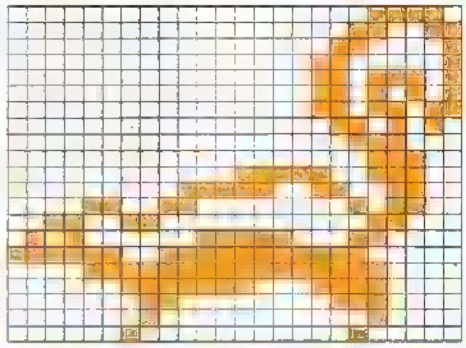


```
0,3,56,36,0,124,8
0,252,56,1,254,92,3
254,252,7,255,254,7,239
31,135,231,31,227,227,15
243,243,15,249,242,63,253
242,39,253,244,33,252,240
3,254,240,7,254,224,7
255,192,7,255,192,0,255
0,3,254,0,15,248,0
```

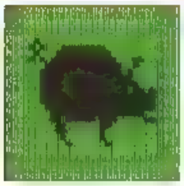


SKUNK

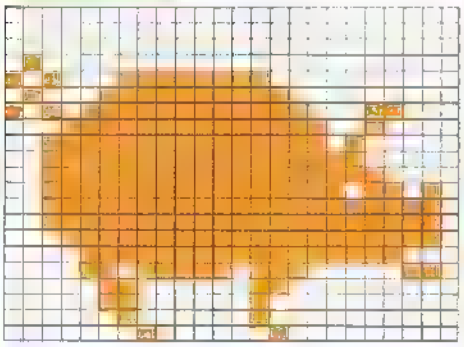
```
0,0,30,0,0,51,0
0,97,0,0,193,0,0
221,0,0,213,0,0,219
0,0,106,0,0,56,0
0,28,0,15,156,0,124
204,12,192,44,51,0,24
80,62,48,252,255,224,15
255,240,3,248,240,1,128
112,1,0,16,2,0,32
```



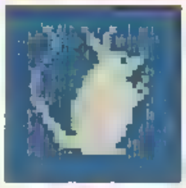
PIG



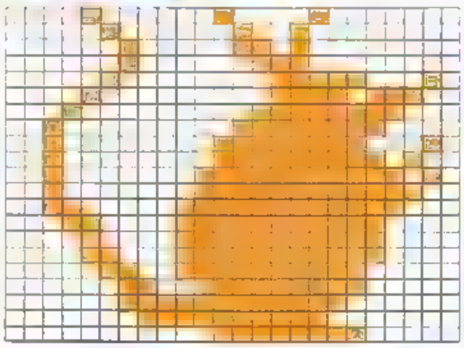
```
0,0,0,0,0,0,0
0,0,64,0,0,163,252
0,71,254,0,175,255,24
31,255,144,63,255,160,63
255,224,63,255,240,63,255
218,63,255,254,63,255,254
31,255,254,15,255,248,13
247,6,6,6,0,6,4
0,2,4,0,1,2,0
```



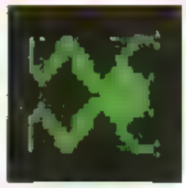
MOUSE



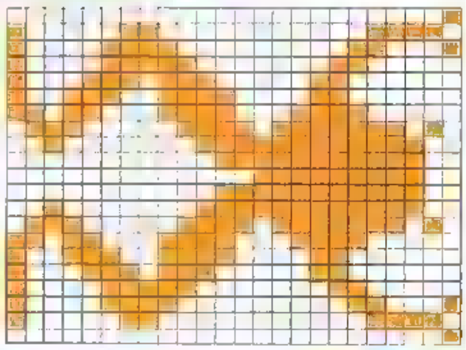
```
8,16,128,4,9,0,2
13,0,2,7,0,4,3
226,8,3,220,16,7,248
32,15,240,32,31,226,32
31,244,32,63,250,32,63
252,48,127,248,24,127,240
8,127,240,32,127,240,6
127,224,3,63,224,3,255
192,1,255,128,0,31,224
```



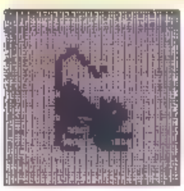
FROG



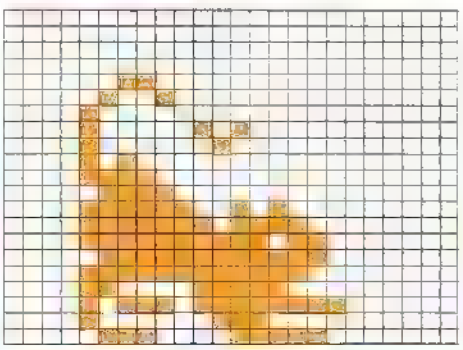
```
0,0,1,131,0,30,1
128,49,135,192,96,1
192,156,240,192,216,24
112,123,250,32,63,252,0
31,252,0,7,252,0,31
252,32,63,252,112,123,25
216,241,224,156,240,192,
224,192,135,192,96,135,1
49,131,0,30,0,0,1
```



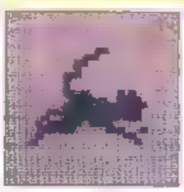
CAT



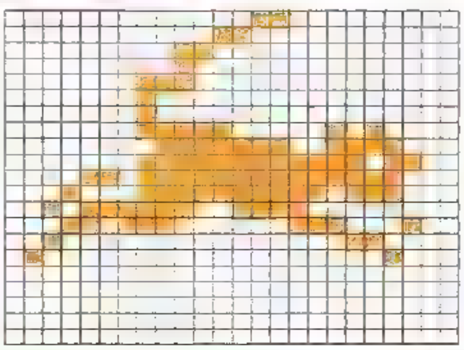
```
0,0,0,0,0,0,0
0,0,0,0,0,3,0
0,4,128,0,8,64,0
8,40,0,8,16,0,10
0,0,15,0,0,7,128
0,15,202,0,15,239,0
15,253,128,7,255,128,15
255,0,12,62,0,11,191
192,8,28,0,7,15,128
```



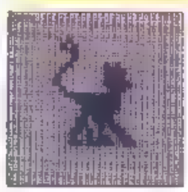
CAT



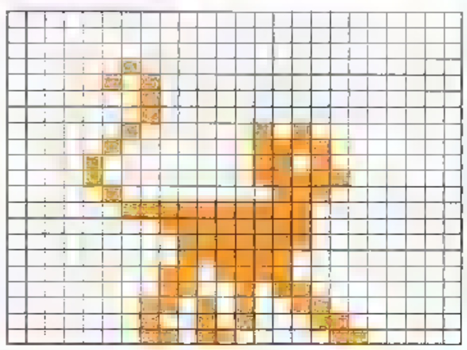
```
0,6,0,0,24,0,0
96,0,0,64,0,1,192
0,1,0,0,1,0,0
1,240,80,0,254,120,1
255,236,13,255,248,17,255
184,47,237,0,27,129,196
32,0,40,64,0,8,0
0,0,0,0,0,0,0
0,0,0,0,0,0,0
```



CAT

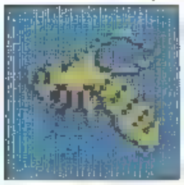


```
0,0,0,0,0,0,0
0,0,2,0,0,5,0
0,1,0,0,1,0,0
2,5,0,4,7,128,8
6,128,8,7,192,4,3
128,3,255,0,0,255,0
0,127,0,0,126,0,0
222,0,0,203,0,1,169
128,1,44,192,1,182,96
```

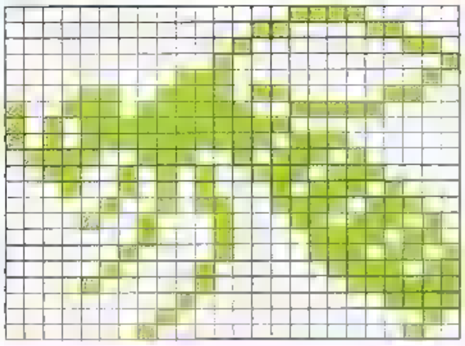


BUGS AND SNAILS

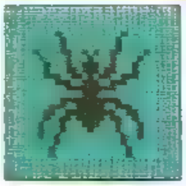
WASP



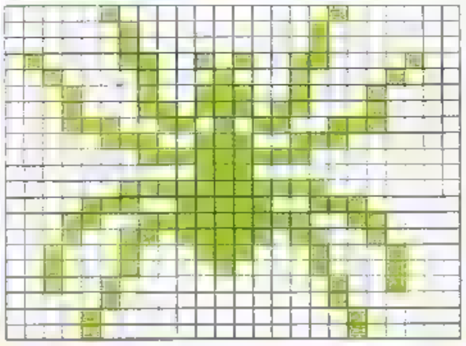
0,1,224,0,6,24,0
8,6,0,16,1,0,112
2,12,246,12,157,249,240
175,254,0,175,253,192,128
203,160,50,167,112,18,162
232,4,179,220,9,17,186
1,16,246,2,16,237,4
32,123,8,32,63,0,64
15,0,128,2,1,0,4



SPIDER

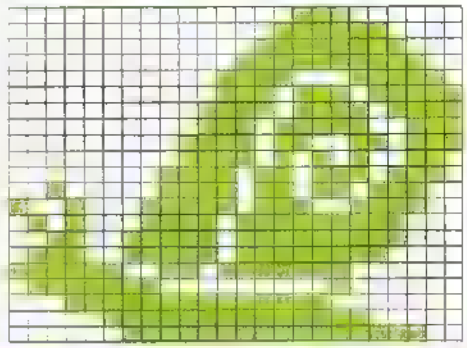


4,0,64,2,0,128,2
0,128,67,41,132,33,17
8,17,57,16,17,187,16
28,214,112,7,57,132,1
255,0,0,56,0,3,255
128,14,124,224,25,255,48
19,125,144,34,56,136,34
16,136,36,0,72,4,0
64,8,0,32,8,0,32



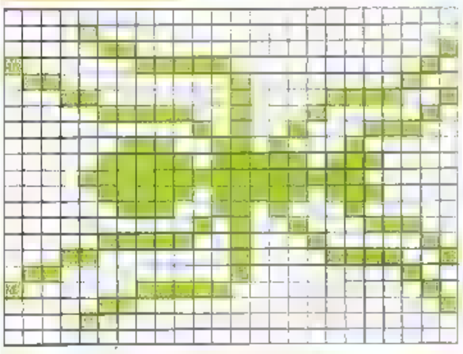
SNAIL

0,1,192,0,3,240,0
7,248,0,15,252,0,30
62,0,29,223,0,61,239
0,59,231,0,123,55,0
122,166,0,246,238,32,246
220,145,247,60,81,239,248
51,239,248,115,239,240,252
223,192,127,0,64,31,255
128,7,255,224,3,255,252



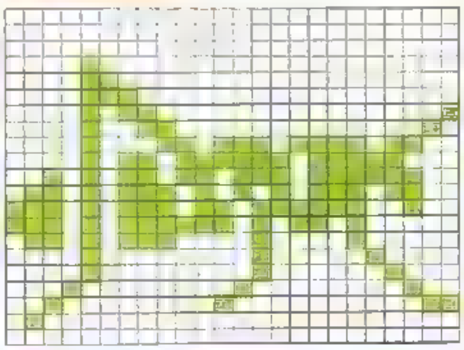
ANT

0,0,0,8,0,0,6
0,1,129,240,2,96,8
4,24,8,121,7,200,130
0,41,28,3,154,32,7
223,112,15,255,224,7,223
112,3,154,32,0,41,28
7,200,130,24,8,121,96
8,4,129,240,2,6,0
1,8,0,0,0,0,0

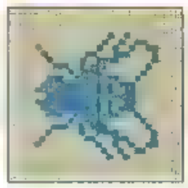


CRICKET

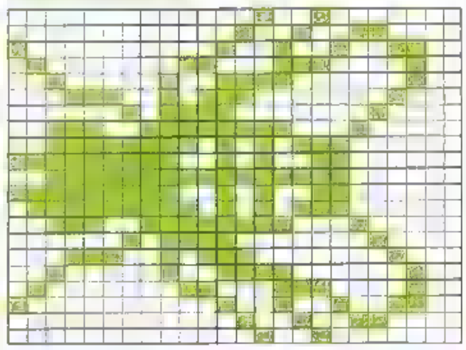
0,0,0,0,0,0,0
0,0,8,0,0,12,0
0,14,0,0,11,0,1
9,128,2,0,221,236,11
110,184,43,110,188,107,186
236,235,183,228,235,245,16
107,132,32,8,4,16,8
4,8,16,8,4,32,16
3,64,0,0,0,0,0



FLY

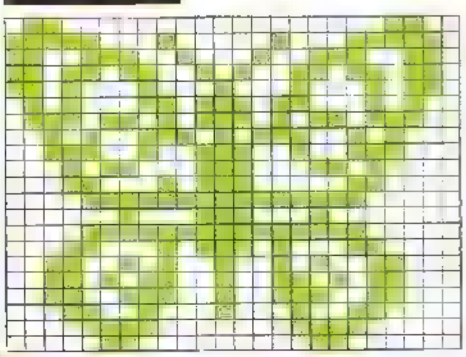


0,4,128,0,9,48,128
18,76,64,99,132,32,158
4,28,184,8,2,240,16
57,254,32,127,213,192,255
148,192,63,255,192,255,148
192,127,213,192,57,254,32
2,240,16,28,184,8,32
158,4,64,99,132,128,18
76,0,9,48,0,4,128

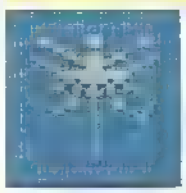


BUTTERFLY

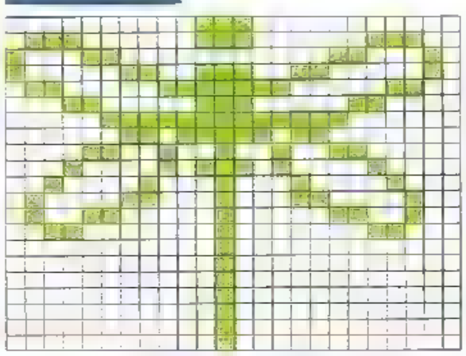
96,0,12,248,130,62,206
68,230,219,41,182,209,17
22,81,187,20,118,186,220
41,215,40,57,57,56,22
56,208,24,186,48,15,255
224,2,56,128,15,255,224
24,186,48,50,146,152,53
147,88,51,147,152,25,17
48,15,1,224,6,0,192



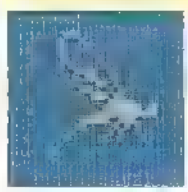
DRAGONFLY



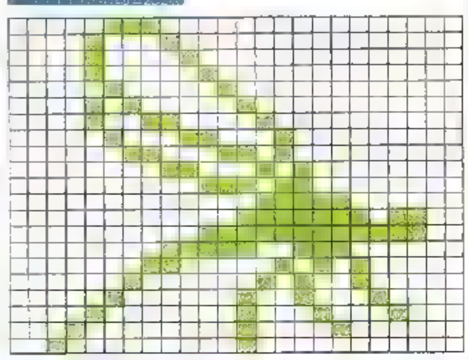
0,56,0,112,56,28,140
0,98,131,57,130,96,254
12,24,56,48,7,255,192
3,125,128,12,146,96,16
186,16,33,17,8,67,17
132,76,16,100,48,16,24
0,16,0,0,16,0,0
16,0,0,16,0,0,16
0,0,16,0,0,16,0



DRAGONFLY



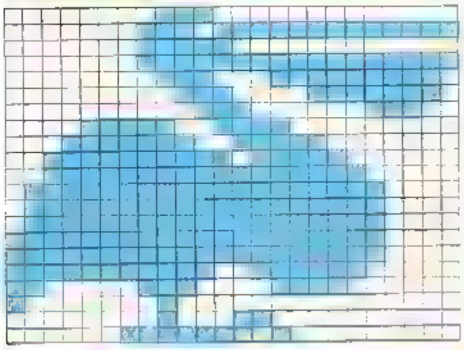
7,0,0,8,128,0,8
64,0,8,32,0,4,16
0,10,8,0,9,132,0
4,98,0,3,26,0,0
197,0,0,59,0,0,7
192,0,15,236,0,63,252
0,249,192,1,130,160,2
5,32,4,9,16,8,8
136,16,8,64,32,8,0



BIRDS

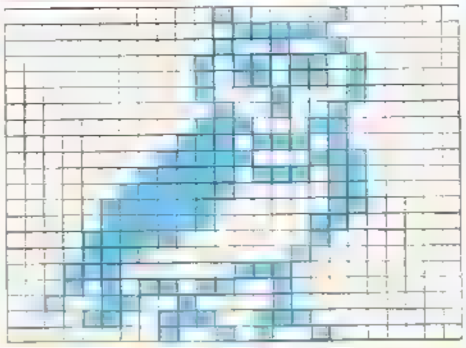
PELICAN

```
15,231,240,30,0,120,31
231,248,92,0,58,92,0
58,92,66,50,76,255,34
167,255,229,147,255,201,143
255,241,71,255,226,63,255
252,7,255,224,63,255,252
67,255,194,141,255,177,144
126,9,160,0,5,160,0
5,16,0,8,8,0,16
```



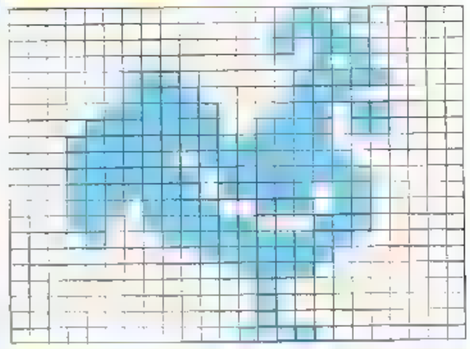
OWL

```
0,0,120,1,5,246,0
1,255,0,1,254,0,3
254,0,3,243,0,7,241
0,15,225,0,63,225,31
255,225,127,255,247,255,255
247,255,255,244,255,255,240
255,255,240,255,253,224,255
249,225,249,243,225,252,3
220,126,3,240,31,243,135
```



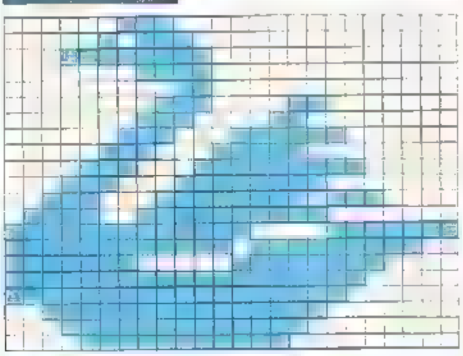
COCKEREL

```
0,0,0,0,0,0,0
0,55,0,0,127,0,0
126,0,0,252,0,1,248
0,3,240,0,15,240,0
127,240,3,255,240,31,255
248,63,255,248,127,255,248
255,255,248,255,255,244,253
255,244,252,56,246,127,0
238,63,129,199,31,193,129
```

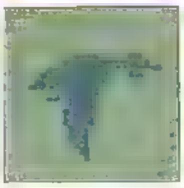


SWAN

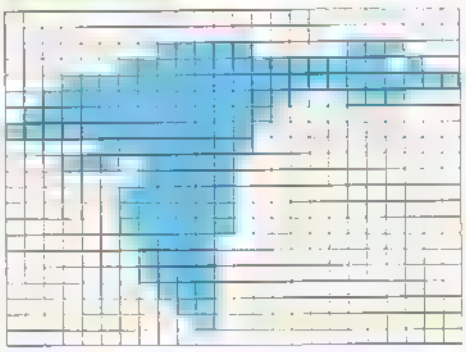
```
0,0,2,0,0,14,0
0,62,0,0,126,0,0
254,0,1,254,[illegible],251
0,255,255,[illegible],255,255,13
250,[illegible],63,250,191,137,250
191,255,[illegible],[illegible],252,254,[illegible]
115,255,255,3,255,254,0
31,240,0,3,240,0,0
240,0,0,48,0,0,[illegible]
```



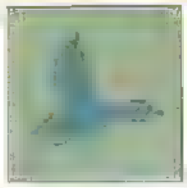
DUCK



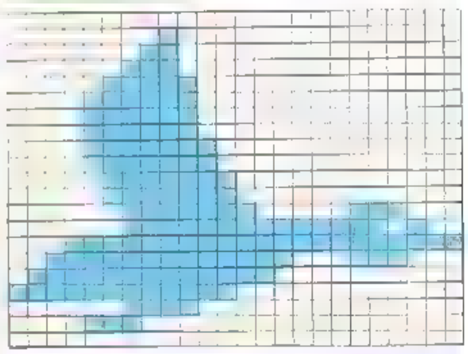
```
[illegible]
[illegible]
12,0,[illegible],28,240,0,54
255,252,[illegible],255,[illegible],240,[illegible]
255,224,255,255,224,255,254
[illegible]
[illegible]
[illegible]
[illegible]
```



DUCK

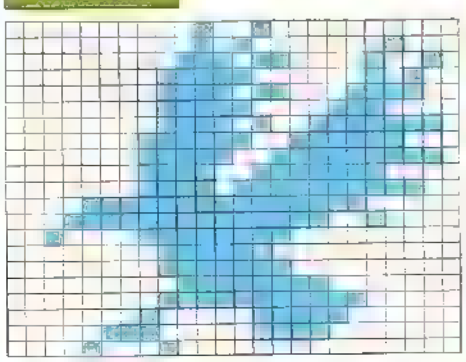


```
0,24,0,0,60,0,0
60,0,0,126,0,0,126
0,0,126,0,0,126,0
220,90,36,78,255,114,195
255,195,255,255,217,166,126
[illegible],161,255,133,156,36,101
[illegible],[illegible],[illegible],164,[illegible],37,180
245,37,146,74,105,81,36
206,8,144,144,4,77,32
```



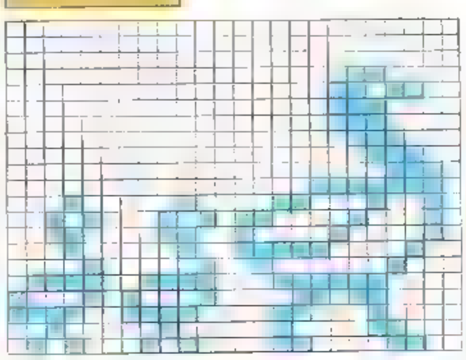
EAGLE

```
0,4,192,0,0,0,0
18,96,0,148,158,1,85
48,2,05,62,95,86,113
224,46,198,240,127,152,127
255,224,127,255,192,127,255
224,240,127,152,224,46,198
98,06,113,2,35,52,1
85,48,0,148,158,0,18
96,0,0,0,0,4,192
```



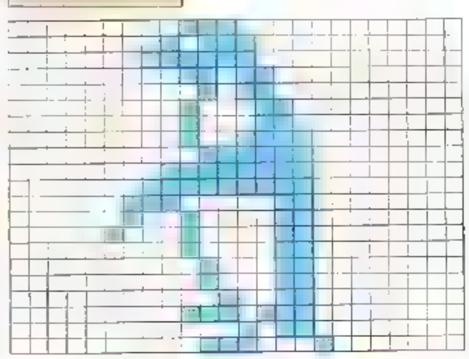
DUCK AND DUCKLINGS

```
0,6,0,0,[illegible],0,1
247,175,[illegible],[illegible],128,15,253
2,31,254,[illegible],[illegible],[illegible],78
126,253,[illegible],119,127,232,63
127,[illegible],233,127,252,126,127
254,124,245,[illegible],57,255,20
31,254,138,15,252,133,3
251,128,0,240,0,0,24
0,0,[illegible],0,0,0,0
```

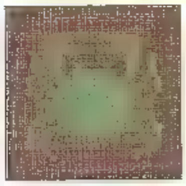


PENGUIN

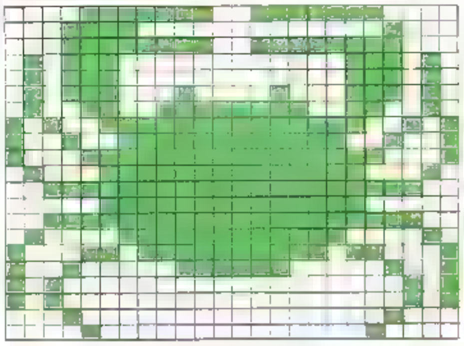
```
3,224,0,6,28,0,11
254,0,23,253,0,47,15
0,94,2,130,124,[illegible],65
248,1,194,216,0,196,232
0,164,244,0,228,122,0
230,63,0,162,31,193,194
15,112,194,7,[illegible],134,2
243,140,0,3,24,0,3
176,0,1,224,0,0,128
```



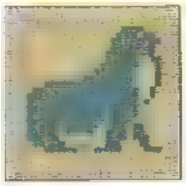
CRAB



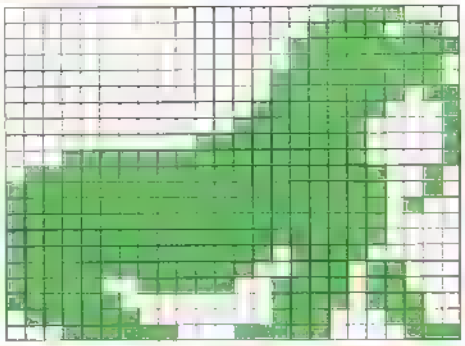
```
0,224,0,1,223,255,1
240,0,1,255,255,1,227
254,0,112,254,0,56,28
15,158,0,31,207,128,63
247,192,63,255,224,127,255
240,127,255,240,127,255,240
255,255,240,255,255,240,255
207,224,195,135,192,129,0
0,1,24,0,7,244,0
```



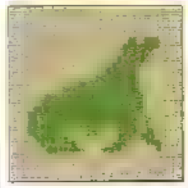
WALRUS



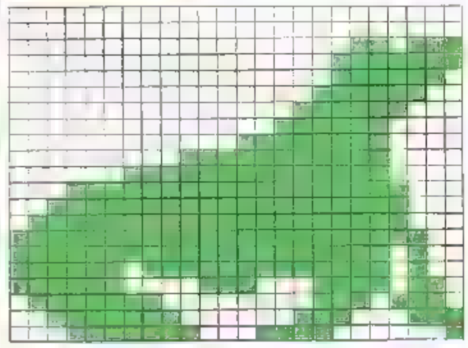
```
0,16,64,0,15,128,0
18,64,0,45,160,0,45
160,0,34,32,0,18,64
0,56,192,0,119,64,0
248,224,1,247,96,3,240
96,7,224,96,7,224,192
15,123,128,14,1,0,28
14,0,21,178,0,38,66
0,12,231,0,0,148,128
```



SEAL

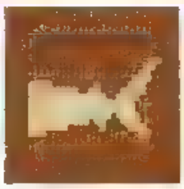


```
0,1,160,0,3,192,0
2,224,0,0,160,3,129
240,3,193,192,7,227,176
15,231,176,31,255,128,31
191,192,31,223,224,31,223
96,29,231,96,25,248,224
8,255,192,0,63,128,0
15,0,0,4,0,0,4
0,0,4,0,0,11,0
```

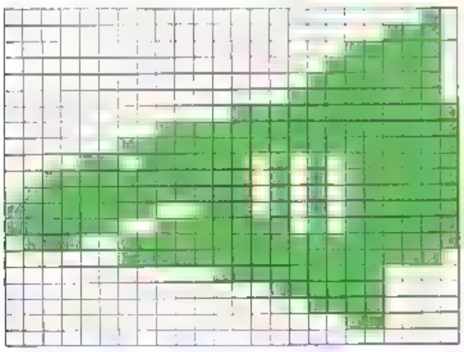


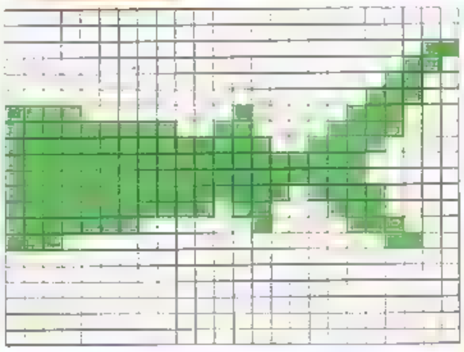
SHARK

```
0,128,0,5,192,0,31
224,0,1,224,0,0,96
0,0,225,128,1,195,0
0,143,192,7,31,0,14
63,224,28,127,128,56,255
112,121,255,128,255,248,127
255,247,252,254,15,240,255
255,224,255,255,192,63,255
128,31,255,0,15,254,0
```



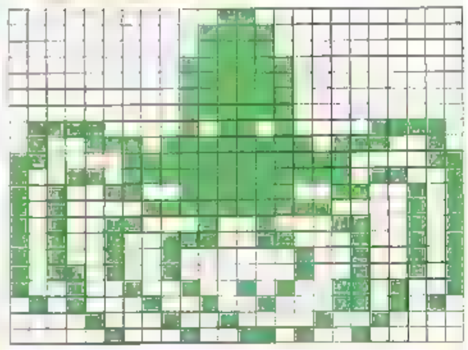
```
0,0,0,0,0,0,0
248,56,3,255,244,31,255
255,63,254,56,127,252,0
255,248,0,7,248,0,25
240,0,1,240,0,3,240
0,3,240,0,3,240,0
1,224,0,1,224,0,1
224,0,0,224,0,0,96
0,0,32,0,0,0,0
```





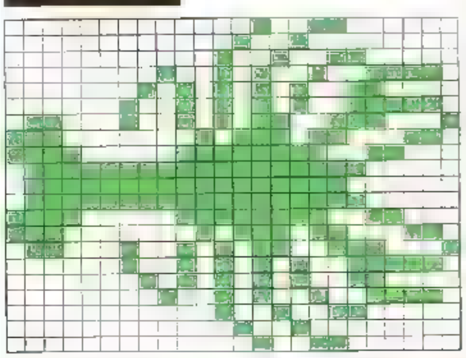
OCTOPUS

```
0,0,0,0,128,0,1
128,0,3,128,0,7,192
0,15,192,0,15,192,0
15,192,0,15,224,0,15
224,0,7,240,0,7,240
0,3,248,56,3,255,244
31,255,255,63,254,56,127
252,0,255,240,0,3,192
0,14,0,0,0,0,0
```

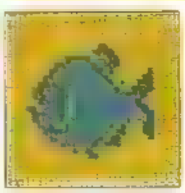


LOBSTER

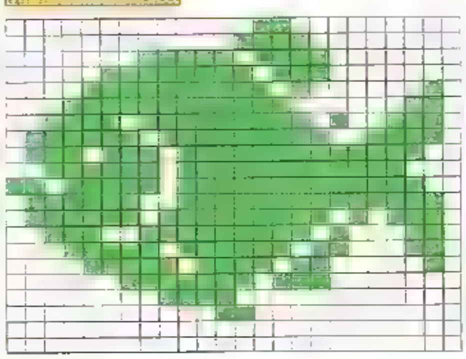
```
0,36,0,0,40,8,0
128,18,0,240,20,0,246
60,0,248,112,0,244,254
1,240,248,1,243,254,1
231,240,1,239,252,15,255
224,31,255,144,33,254,0
0,252,0,0,254,0,0
127,128,0,255,224,1,7
192,7,7,128,8,129,7
```



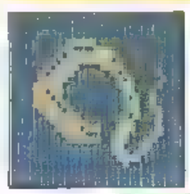
PIRANHA



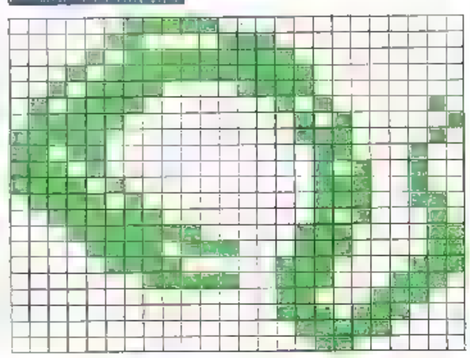
```
0,0,0,0,0,0,0
0,0,0,0,48,0,0
103,0,0,112,0,0,112
0,0,56,0,0,28,0
0,14,0,0,254,15,67
18,40,172,34,48,200,66
16,79,132,8,36,8,121
227,240,138,33,176,243,195
16,32,130,24,81,67,12
```



MORAY EEL

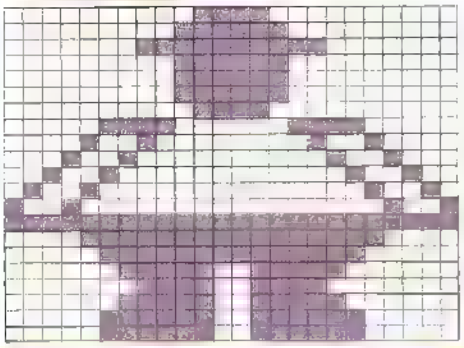


```
0,112,0,0,184,0,1
248,0,0,108,0,0,44
0,0,70,0,0,70,0
0,79,0,0,63,0,0
255,0,1,255,0,0,131
0,2,67,0,4,67,0
0,67,0,0,35,0,0
35,0,0,19,0,0,119
0,0,11,0,0,56,128
```



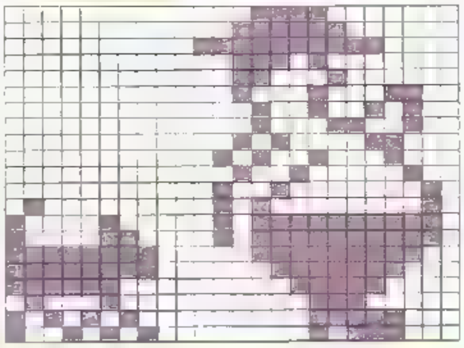
SHERRIFF

```
0,60,0,0,126,0,1
255,128,0,126,0,0,126
0,0,126,0,0,60,0
7,129,224,9,0,144,18
0,72,36,0,36,72,0
18,144,0,9,255,255,255
15,255,240,7,255,224,3
231,192,1,231,128,3,231
192,7,231,224,7,231,224
```



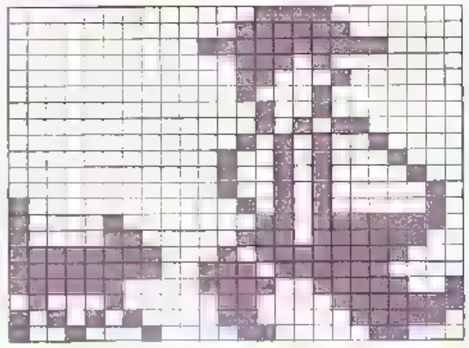
SHERRIFF

```
3,7,128,0,15,192,0
63,240,0,14,128,0,12
64,0,8,140,0,4,148
3,4,130,0,10,24,0
20,136,0,9,4,0,18
2,64,20,2,137,23,254
134,23,30,255,7,252,255
3,248,252,1,240,68,0
224,170,0,248,145,0,184
```

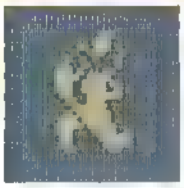


SHERRIFF

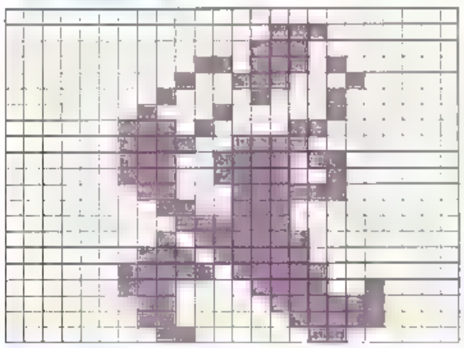
```
0,7,128,0,15,192,0
63,240,0,14,128,0,12
64,0,8,128,0,4,128
0,5,96,0,10,144,0
18,136,0,18,132,0,18
130,128,10,130,132,6,254
134,14,252,255,23,248,255
59,240,252,60,243,68,24
127,170,24,62,145,12,24
```



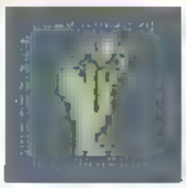
HUNCHBACK



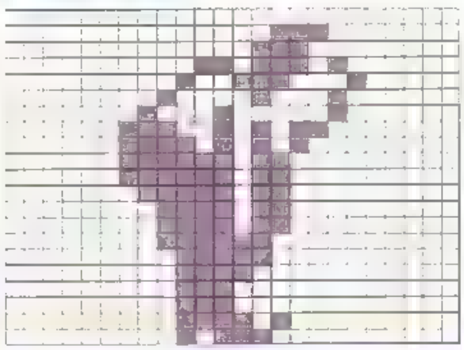
```
0,0,0,0,3,128,0
7,0,0,55,64,0,78
32,0,132,64,1,16,64
3,161,128,3,207,0,3
158,192,3,158,64,1,143
192,0,206,0,0,126,0
0,191,0,1,223,0,3
239,128,3,135,144,1,131
240,1,129,240,0,192,192
```



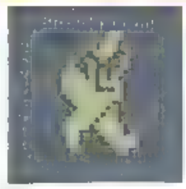
HUNCHBACK



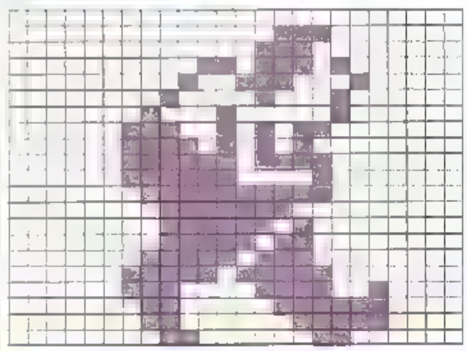
```
0,0,0,0,3,128,0
7,0,0,55,64,0,78
32,0,132,64,1,16,64
3,149,128,3,213,0,3
246,0,3,246,0,1,246
0,0,246,0,0,246,0
0,252,0,0,124,0,0
120,0,0,112,0,0,112
0,0,112,0,0,92,0
```



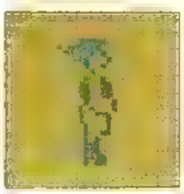
HUNCHBACK



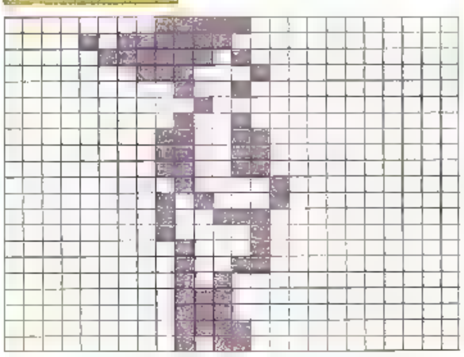
```
0,0,0,0,3,129,0
7,0,0,55,64,0,78
32,0,132,64,1,16,64
3,149,128,3,213,0,3
247,128,3,240,128,1,255
128,0,254,0,0,125,0
0,251,0,1,247,0,3
239,128,3,135,144,1,131
240,1,129,240,0,192,192
```



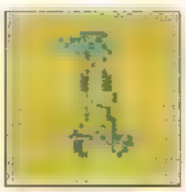
DWARF



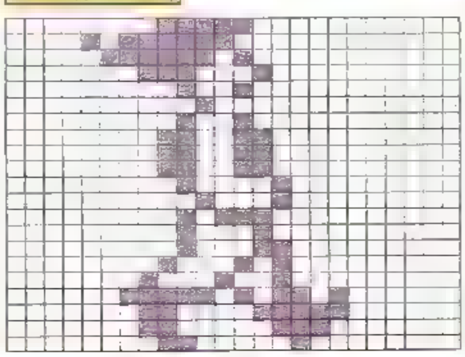
```
0,248,0,11,240,0,7
232,0,1,196,0,0,72
0,0,32,0,0,72,0
0,204,0,0,204,0,0
204,0,0,66,0,0,162
0,0,156,0,0,132,0
0,60,0,0,76,0,0
80,0,0,112,0,0,112
0,0,120,0,0,88,0
```



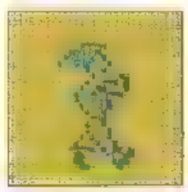
DWARF



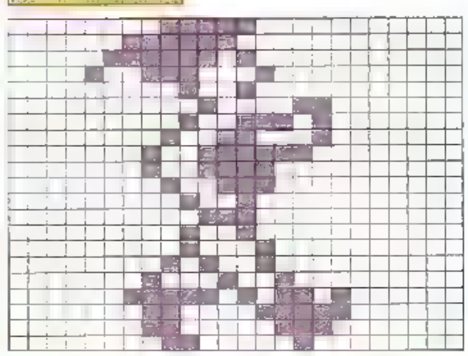
```
0,248,0,11,240,0,7
232,0,1,196,0,0,72
0,0,32,0,0,72,0
0,204,0,0,204,0,0
204,0,0,66,0,0,34
0,0,92,0,0,68,0
0,70,0,0,138,0,1
18,0,3,235,64,1,135
192,1,131,128,0,193,0
```



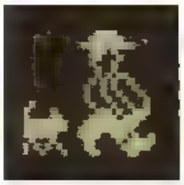
DWARF



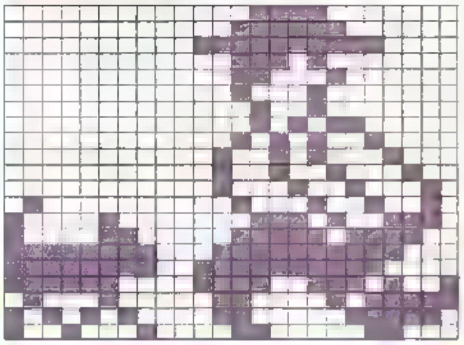
```
0,248,0,3,240,0,7
232,0,9,196,0,0,72
0,0,33,128,0,78,128
0,152,128,1,63,0,1
60,0,0,156,0,0,72
0,0,56,0,0,72,0
0,68,0,0,164,0,1
18,0,3,235,64,1,135
192,1,131,128,0,193,0
```



SHERRIFF

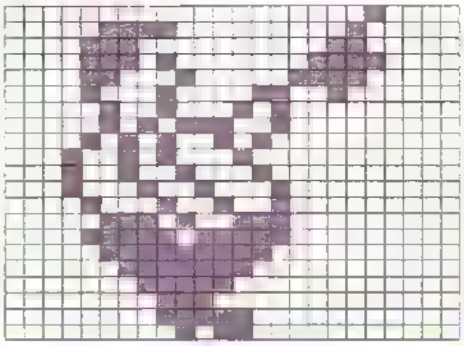


```
0,7,128,0,15,192,0
63,240,0,14,128,0,12
64,0,8,128,0,4,128
0,5,96,0,10,144,0
18,136,0,18,68,0,17
34,64,8,146,132,7,78
134,15,188,255,31,248,255
63,240,252,60,231,68,24
95,170,24,62,145,12,24
```



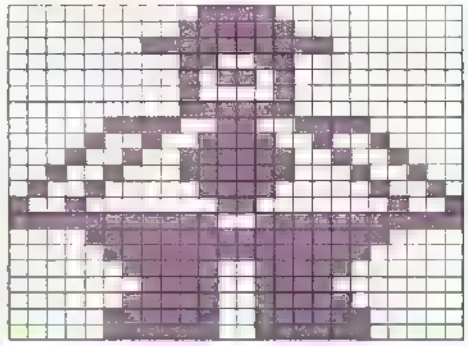
SHERRIFF

```
0,0,15,7,128,32,14
0,96,14,128,240,12,65
224,8,134,64,4,138,0
5,114,0,10,148,0,18
136,0,18,68,0,17,34
0,8,154,0,7,78,0
15,195,0,7,252,0,3
248,0,1,240,0,0,224
0,0,248,0,0,184,0
```



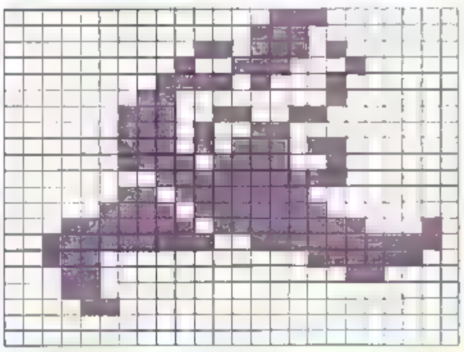
SHERRIFF

```
0,60,0,0,126,0,1
255,128,0,102,0,0,90
0,0,66,0,0,126,0
7,153,224,9,60,144,18
126,72,36,126,36,72,60
18,144,24,9,255,231,255
15,255,240,7,255,224,3
231,192,1,231,128,3,231
192,7,231,224,7,231,224
```



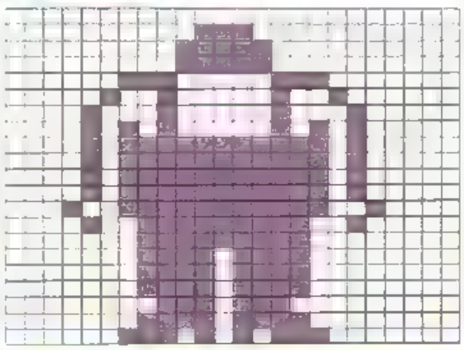
HUNCHBACK

```
0,3,128,0,7,0,0
55,64,0,78,32,0,244
64,1,128,64,3,153,128
3,166,0,3,174,192,3
222,64,0,255,192,3,127
0,7,191,128,31,223,226
63,231,254,56,0,252,24
0,48,24,0,0,12,0
0,0,0,0,0,0,0
```

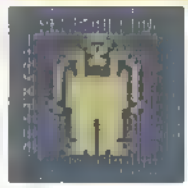


HUNCHBACK

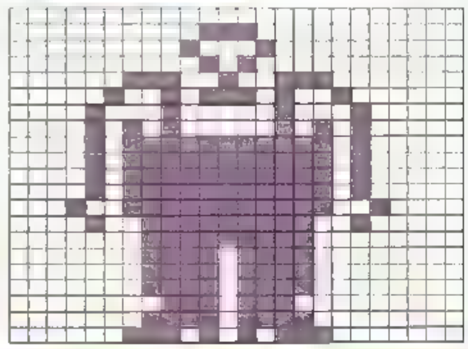
```
0,0,0,0,55,0,0
124,0,0,124,0,3,131
128,4,130,64,8,130,32
10,130,160,11,255,160,11
255,160,11,255,160,11,255
160,19,255,144,9,255,32
1,255,0,1,239,0,1
239,0,1,239,0,0,238
0,1,239,0,3,171,128
```



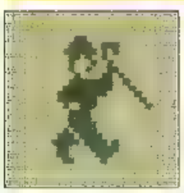
HUNCHBACK



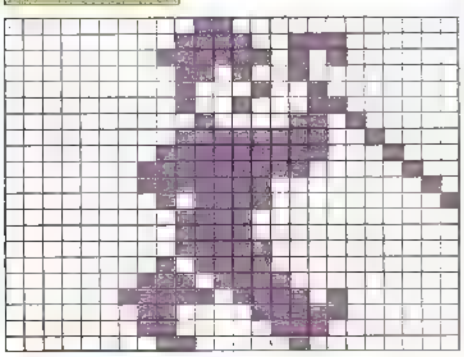
```
0,0,0,0,56,0,0
68,0,0,40,0,3,147
128,4,186,64,8,130,32
10,130,160,11,255,160,11
255,160,11,255,160,11,255
160,19,255,144,9,255,32
1,255,0,1,239,0,1
239,0,1,239,0,0,238
0,1,239,0,3,171,128
```



CHARLIE CHAPLIN

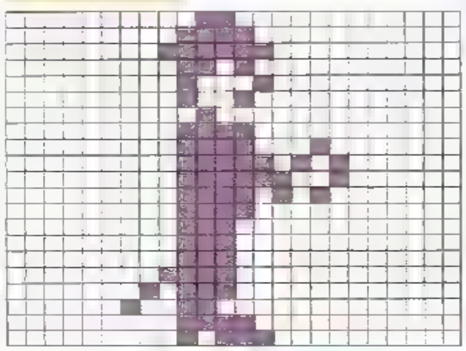


```
0,48,0,0,121,152,0
253,64,0,105,0,0,68
128,0,72,192,0,33,160
0,127,144,0,255,136,1
255,4,1,252,2,0,156
1,0,120,0,0,120,0
0,124,0,0,188,0,1
222,0,3,239,64,1,135
192,1,131,128,0,193,0
```



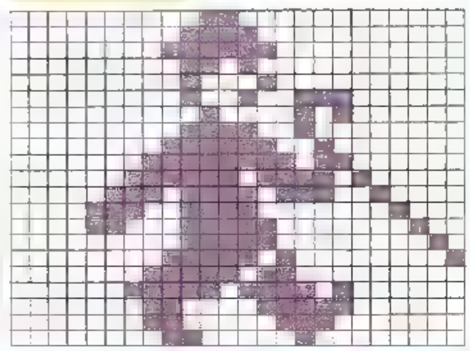
CHARLIE CHAPLIN

```
0,48,0,0,120,0,0
252,0,0,104,0,0,68
0,0,72,0,0,32,0
0,120,0,0,120,128,0
125,64,0,126,64,0,122
128,0,120,0,0,112,0
0,112,0,0,112,0,0
240,0,1,112,0,2,96
0,0,120,0,0,92,0
```



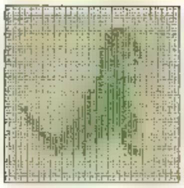
CHARLIE CHAPLIN

```
0,48,0,0,120,0,0
252,0,0,104,0,0,68
0,0,73,192,0,33,64
0,121,0,0,252,128,1
254,192,3,247,160,7,123
144,6,120,8,0,124,4
0,124,2,0,250,1,1
246,0,3,239,64,1,135
192,1,131,128,0,193,0
```

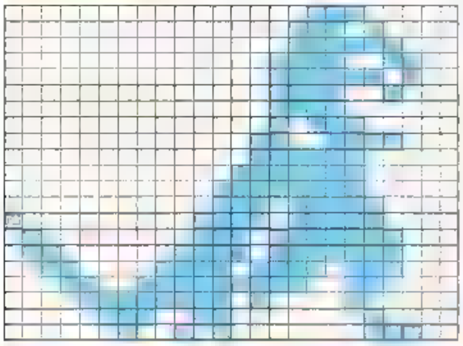


DINOSAURS

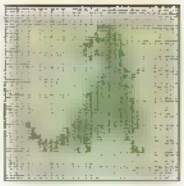
TYRANNOSAURUS



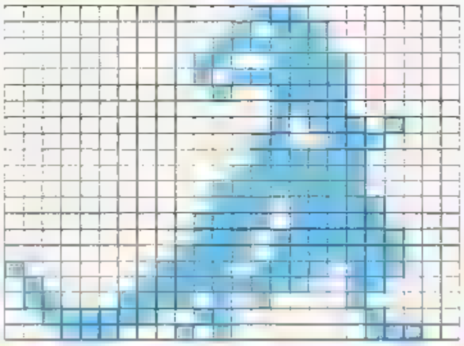
```
0,0,224,0,1,208,0
1,248,0,3,204,0,3
244,0,3,200,0,3,192
0,2,224,0,6,120,0
7,224,0,15,224,0,31
224,0,29,240,128,61,240
64,59,248,96,123,248,48
247,184,25,255,48,15,244
32,7,48,16,2,80,29
```



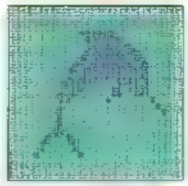
TYRANNOSAURUS



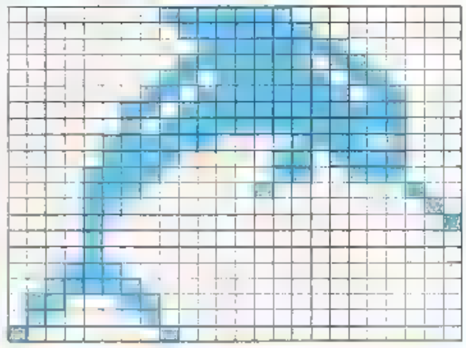
```
0,7,1,0,11,128,0
31,128,0,51,192,0,47
192,0,19,192,0,3,192
0,2,230,[illegible],112,0
[illegible]
224,0,29,240,0,61,240
0,59,248,0,123,248,128
247,184,25,255,48,99,244
32,63,48,16,28,80,28
```



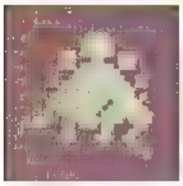
ICHTHYOSAURUS



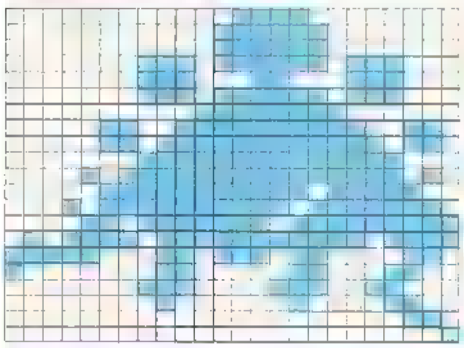
```
0,254,0,0,127,0,0
63,192,0,127,96,0,223
176,1,191,216,3,127,248
2,255,248,7,225,232,7
131,120,15,3,56,14,4
3,12,0,2,8,0,1
5,0,3,8,0,0,28
2,0,52,0,0,39,0
0,65,0,0,128,128,0
```



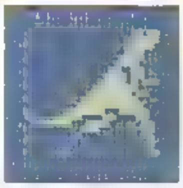
STEGOSAURUS



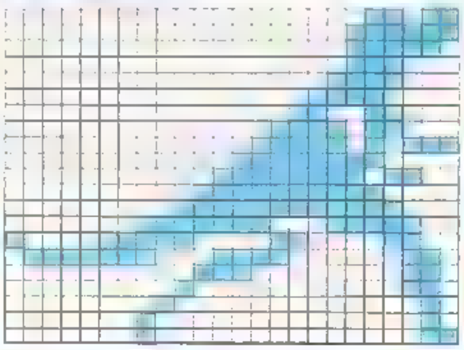
```
0,15,0,0,31,128,0
31,128,1,225,184,1,128
56,1,223,184,[illegible]
0,127,236,0,255,246,1
255,248,1[illegible],255,250,7,255
124,33,254,254,[illegible]
94,255,239,248,221,124,128
193,140,1,195,132,0,129
12,0,0,6,0,0,1
```



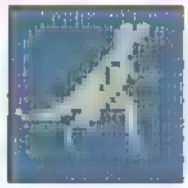
ALLOSAURUS



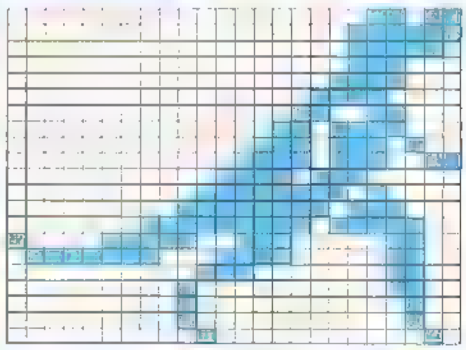
```
[illegible]
[illegible]
[illegible]
[illegible]
[illegible]
240,[illegible],124
[illegible]
[illegible]
[illegible]
```



ALLOSAURUS



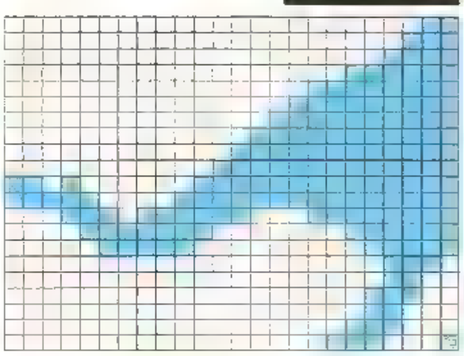
```
0,0,27,0,0,63,0
0,62,0,0,[illegible],0,0
[illegible],240,0,1,192
0,3,120,0,7,116,0
[illegible],1,5,0,30,224,0,6[illegible]
[illegible],8,0,127,184,1,255,124
[illegible],236,12,127,30,[illegible],0
56,12,0,64,12,0,64
4,0,14,4,0,17,2
```

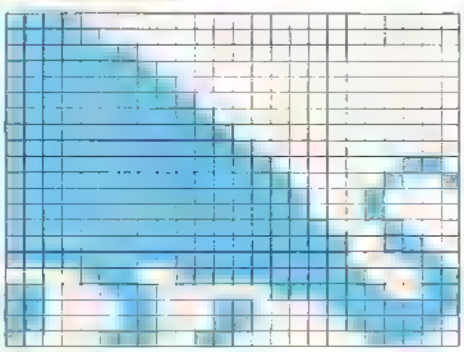


BRONTOSAURUS

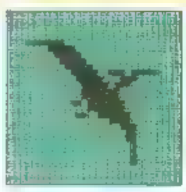
```
0,0,3,0,0,7,0
0,15,0,0,63,0,0
255,0,1,255,0,3,255
0,7,255,0,15,255,0
31,255,208,63,255,248,127
255,29,248,255,15,274,63
7,192,31,0,0,14,0
0,12,0,0,28,0,0
24,0,0,56,0,0,113
```

```
240,0,0,145,0,0,254
0,0,[illegible],128
[illegible],255,[illegible],255,224,0
255,240,0,255,[illegible],0,255
252,6,[illegible],254,[illegible],255,255
[illegible],255,255,[illegible],255,255,136
255,255,230,[illegible],255,198,[illegible]
127,227,139,3,[illegible],195,24
127,195,24,62,137,48,28
```

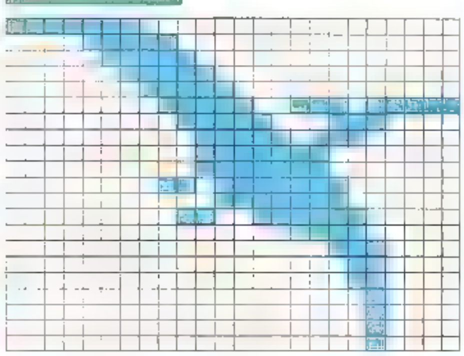




PTERANODON

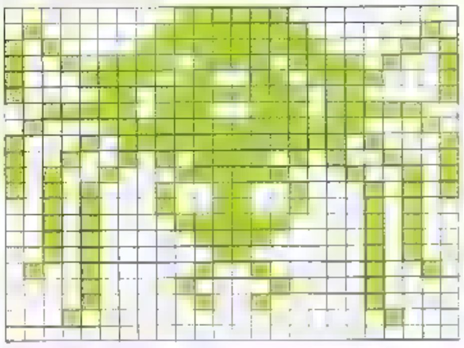


```
254,0,0,15,128,0,7
192,0,3,224,0,1,240
0,0,249,223,0,124,112
0,64,224,0,63,128,0
63,128,0,223,192,0,31
192,0,131,224,0,1,224
0,0,112,0,0,48,0
0,40,0,0,16,0,0
16,0,0,16,0,0,16
```

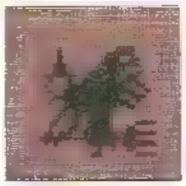


SPIDER

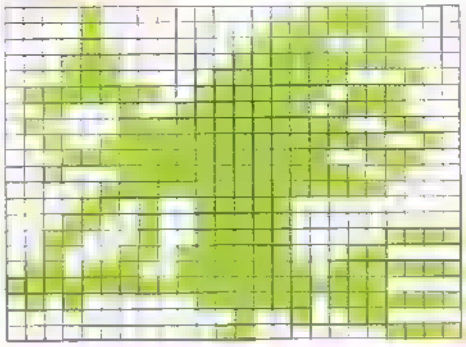
```
0,126,[illegible],54,15[illegible],[illegible],161
255,113,15[illegible],[illegible],[illegible],143,102
24[illegible],134,255,97,62,231,12[illegible]
[illegible],255,226,13[illegible],[illegible],209,146
19[illegible],73,164,12[illegible],[illegible],15[illegible],153
[illegible],45,151,20,10,126,20
4[illegible],[illegible],2[illegible],4[illegible],24,20,72
36,[illegible],[illegible],[illegible],16,[illegible],[illegible]
16,1[illegible],[illegible],8,0,[illegible],[illegible]
```



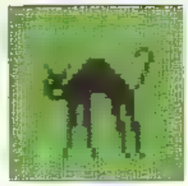
WITCH



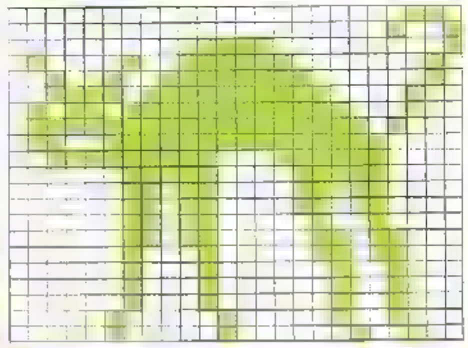
```
8,3,240,8,3,240,8
7,152,28,15,248,28,31
199,127,63,184,34,127,120
6[illegible],255,20[illegible],39,255,24[illegible],31
[illegible]55,156,39,255,232,7,255
240,13,254,[illegible],25,62,0
49,62,[illegible],[illegible],126,48,110
12[illegible],127,255,255,240,80,63
127,0,9,48,0,24,15
```



BLACK CAT

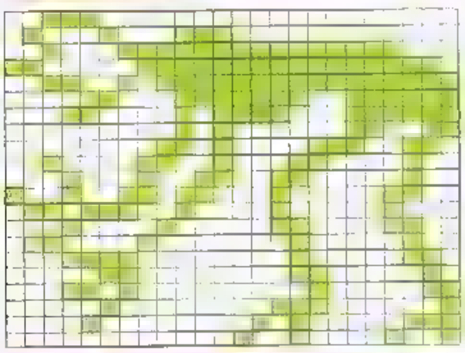


```
0,0,6,0,0,9,0
60,1,66,126,2,36,255
2,61,255,232,91,255,196
127,255,232,103,255,240,63
227,240,3,225,240,3,97
176,3,97,176,3,96,144
3,96,216,2,32,216,2
32,72,2,32,72,2,32
72,4,16,72,4,16,36
```



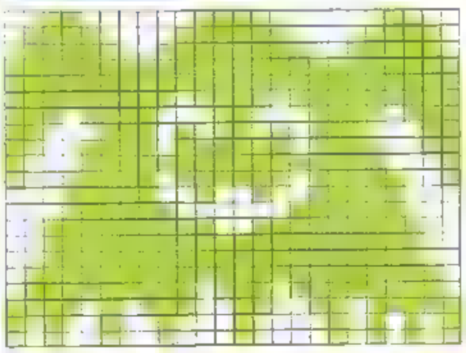
SPECTRE

```
5[illegible],[illegible],[illegible],48,[illegible],0,3[illegible]
[illegible],248,1[illegible],[illegible],[illegible],[illegible],25[illegible]
255,15,1[illegible]7,63,24,[illegible],63
[illegible],8[illegible],8[illegible],[illegible],[illegible],[illegible],3[illegible]
[illegible],1[illegible],1[illegible],[illegible],[illegible],[illegible],1[illegible]
[illegible],1[illegible],6[illegible],[illegible],16,131,12
[illegible],2,4,[illegible],[illegible],6
[illegible],26,[illegible],1[illegible],4,[illegible]
[illegible],8,[illegible],[illegible],[illegible],[illegible]
```



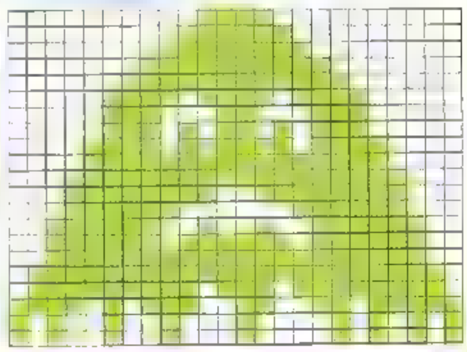
SPOOK

```
4[illegible],[illegible],1[illegible],12[illegible],1[illegible],[illegible],124
25[illegible],[illegible],2[illegible],25[illegible],2[illegible]5,2[illegible]5,25[illegible]
[illegible],[illegible]55,[illegible]6[illegible],2[illegible]5,[illegible],24,247
[illegible],[illegible]0,[illegible]27,[illegible]07,1[illegible]4,2[illegible]3,14[illegible]
[illegible],[illegible],[illegible],[illegible],[illegible],[illegible],[illegible]6
2[illegible],3[illegible],1[illegible]9,24[illegible],6[illegible],231,252
[illegible],2[illegible]5,25[illegible],[illegible]3,[illegible]54,[illegible]52,12[illegible]
[illegible],[illegible],[illegible],[illegible],[illegible]54,[illegible],204
1[illegible],164,136,8,[illegible],164,136,8[illegible]
```



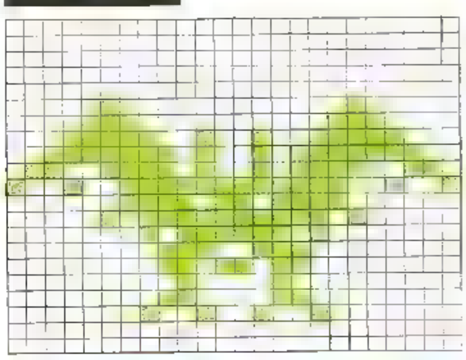
SPOOK

```
0,60,0,0,126,0,0
255,[illegible],1,255,128,3,255
[illegible],7,180,224,7,24,224
7,90,224,15,126,240,15
255,240,31,255,248,31,231
248,31,120,248,63,36,252
63,[illegible],[illegible],63,255,252,127
255,254,127,221,254,245,204
227,164,136,85,164,136,85
```



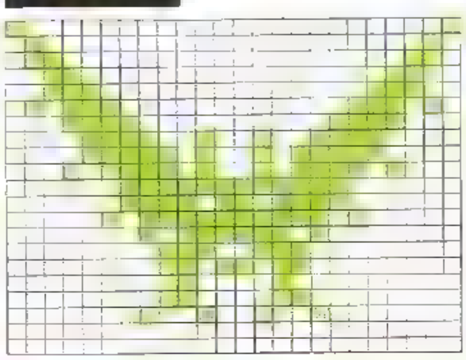
BAT

```
0,[illegible],[illegible],[illegible],[illegible],[illegible],[illegible]
0,0,[illegible],[illegible],[illegible],[illegible],0
[illegible],4,[illegible],[illegible],[illegible]4,0,112
[illegible],3[illegible],24[illegible],[illegible],16[illegible],[illegible],[illegible]
1[illegible],[illegible]3[illegible],[illegible],2[illegible],[illegible],[illegible]
192,[illegible],255,[illegible],[illegible],12[illegible],12[illegible]
[illegible],231,0,[illegible],219,[illegible],[illegible]
[illegible]6,0,[illegible],1[illegible],[illegible],1,[illegible]
128,[illegible],0,[illegible],0,[illegible],[illegible]
```



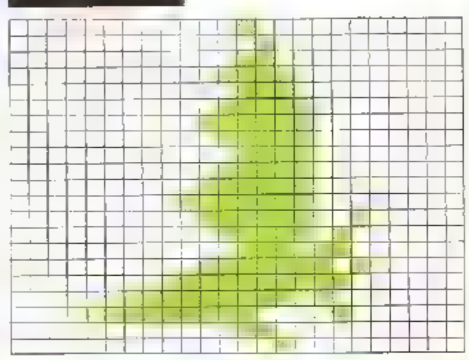
BAT

```
[illegible],[illegible],[illegible],64,0,[illegible],[illegible]6
[illegible],6,4[illegible],0,12,56,0
[illegible],[illegible],[illegible],58,3[illegible],0,[illegible]2[illegible]
[illegible],[illegible]6,2[illegible],1[illegible],1[illegible],14[illegible],[illegible]
[illegible]5,232,[illegible],255,[illegible]2,[illegible],2[illegible]
[illegible],[illegible],[illegible],16[illegible],[illegible],126,125
[illegible],2,[illegible],[illegible],[illegible],[illegible]1[illegible],[illegible],0
6[illegible],[illegible],0,[illegible]55,0,[illegible],[illegible]6
[illegible]28,0,0,0,0,0,0
```



BAT

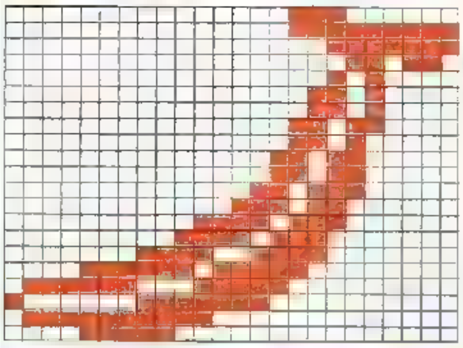
```
0,8,0,0,[illegible],[illegible],0
[illegible]2,[illegible],0,3[illegible],[illegible],0,[illegible]
0,0,31,0,0,63,0
0,31,0,0,15,0,0
31,0,0,63,0,0,127
0,0,63,32,0,31,64
0,14,192,0,63,150,0
255,192,3,254,128,15,246
64,0,12,0,0,2,0
```



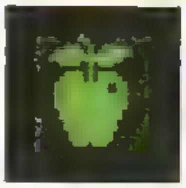
GAMES SYMBOLS

BANANA

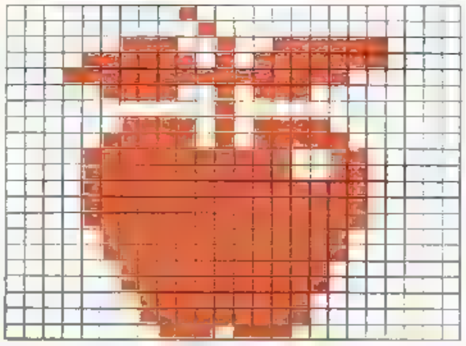
```
0,1,192,0,1,255,0
0,127,0,0,54,0,0
80,0,0,208,0,0,206
0,1,176,0,1,160,0
3,96,0,3,96,0,6
224,0,14,192,0,61,192
0,123,128,1,231,128,15
223,0,126,62,0,129,252
0,127,224,0,15,126,0
```



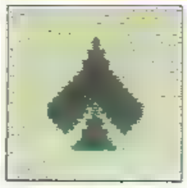
APPLE



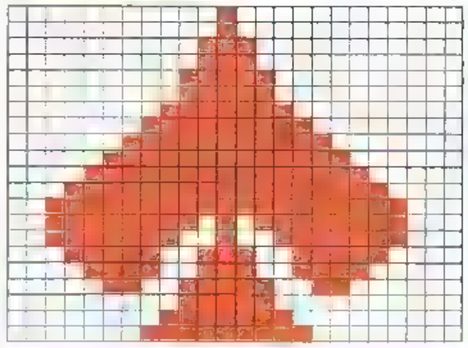
```
0,64,0,0,32,0,3
147,224,15,215,240,31,255
192,7,151,0,0,16,0
3,215,128,7,215,192,15
254,96,15,254,96,15,255
224,15,255,224,15,255,224
7,255,192,7,255,192,3
255,128,3,255,128,1,255
0,1,255,0,0,238,0
```



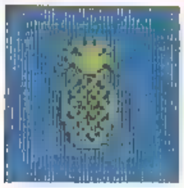
SPADE



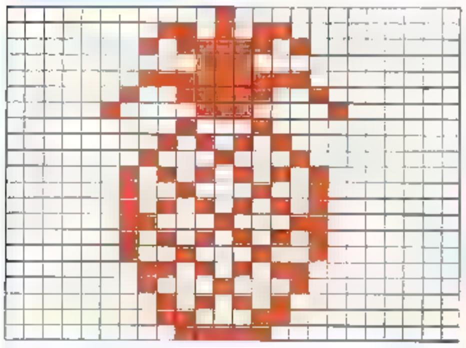
```
0,16,0,0,16,0,0
56,0,0,56,0,0,124
0,0,124,0,0,254,0
1,255,0,3,255,128,7
255,192,15,255,224,31,255
240,63,255,248,63,215,248
63,147,248,31,57,240,14
56,224,4,124,64,0,124
0,0,254,0,1,255,0
```



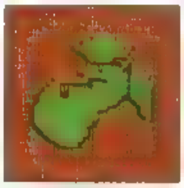
PINEAPPLE



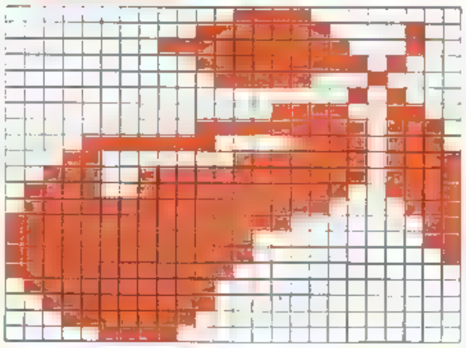
```
0,16,0,0,214,0,1
125,0,0,56,0,1,255
0,2,124,128,4,56,64
0,68,0,0,170,0,1
17,0,2,170,128,2,68
128,2,170,128,3,17,128
2,170,128,2,68,128,1
171,0,1,17,0,0,170
0,0,198,0,0,124,0
```



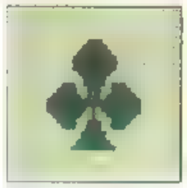
PEAR



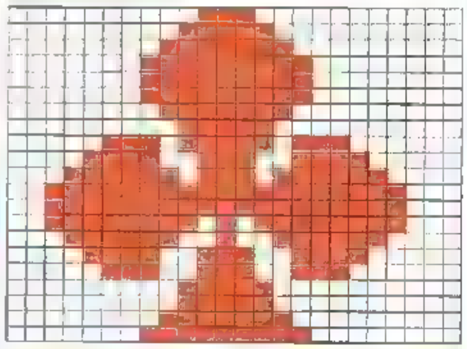
```
0,15,0,0,63,132,0
255,196,0,63,232,0,31
16,0,0,40,0,3,204
0,63,238,15,225,238,24
15,239,57,255,239,121,255
207,127,255,135,255,254,3
255,248,3,255,248,1,255
240,0,127,224,0,63,224
0,31,192,0,7,128,0
```



CLUB

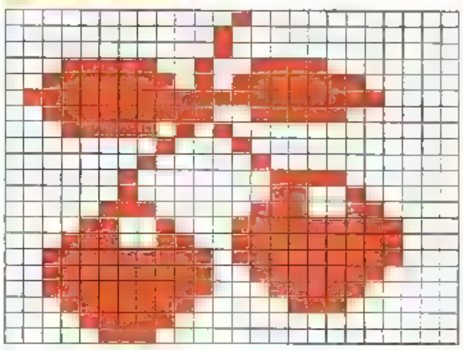


```
0,56,0,0,124,0,0
254,0,1,255,0,1,255
0,1,255,0,0,254,0
0,124,0,7,125,192,15
57,224,31,187,240,63,255
248,63,255,248,63,215,248
31,147,240,15,57,224,7
57,192,0,124,0,0,124
0,0,254,0,1,255,0
```

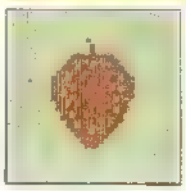


CHERRIES

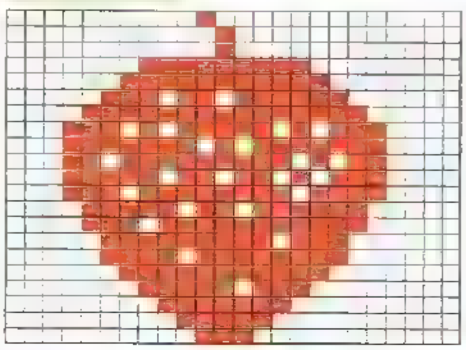
```
0,8,0,0,16,0,0
16,0,31,39,128,63,175
224,127,255,240,63,167,192
31,80,0,0,136,0,1
4,0,2,3,192,2,3
32,7,7,48,12,143,248
28,207,248,63,239,248,63
231,240,63,227,224,31,193
192,15,128,0,7,0,0
```



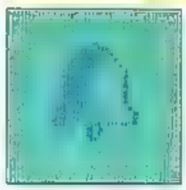
STRAWBERRY



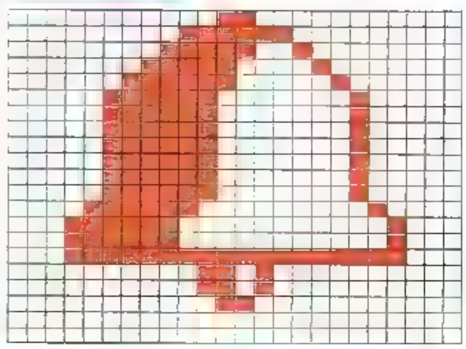
```
0,32,0,0,16,0,0
16,0,1,255,0,3,255
128,7,111,192,15,255,224
29,189,176,31,239,240,30
254,176,27,219,240,31,255
112,13,111,224,15,251,96
5,223,192,7,251,192,3
127,128,1,239,0,0,254
0,0,124,0,0,56,0
```



BELL

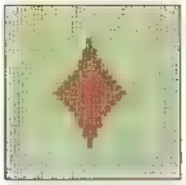


```
0,24,0,0,126,0,0
249,0,1,240,128,3,240
64,7,224,32,7,224,32
7,224,32,7,224,32,7
224,32,7,224,32,7,224
32,15,192,16,31,128,8
31,128,8,31,255,248,0
52,0,0,52,0,0,24
0,0,0,0,0,0,0
```

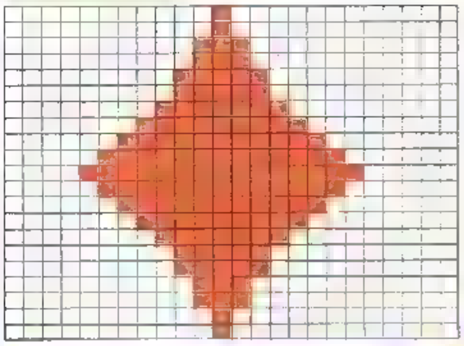


GAMES SYMBOLS

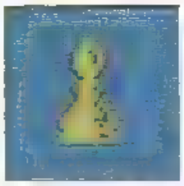
DIAMOND



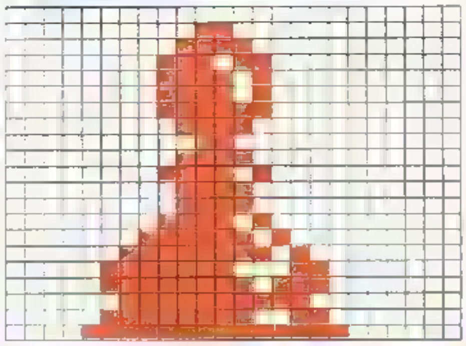
```
0,16,0,0,16,0,0
56,0,0,56,0,0,124
0,0,124,0,0,254,0
1,255,0,3,255,128,7
255,192,15,255,224,7,255
192,3,255,128,1,255,0
0,254,0,0,124,0,0
124,0,0,56,0,0,56
0,0,16,0,0,16,0
```



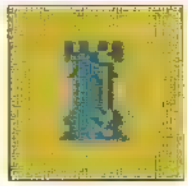
PAWN



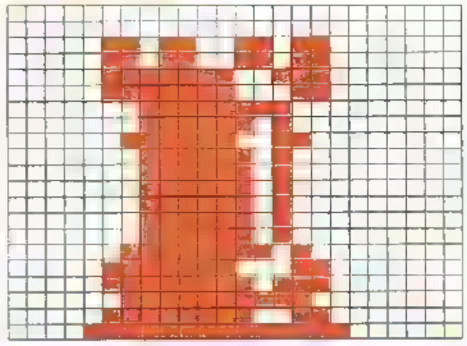
```
0,0,0,0,48,0,0
120,0,0,236,0,0,244
0,0,244,0,0,252,0
0,120,0,0,48,0,0
120,0,0,244,0,0,120
0,0,120,0,0,244,0
1,250,0,3,253,0,7
241,128,7,251,128,3,255
0,7,249,128,15,255,192
```



ROOK

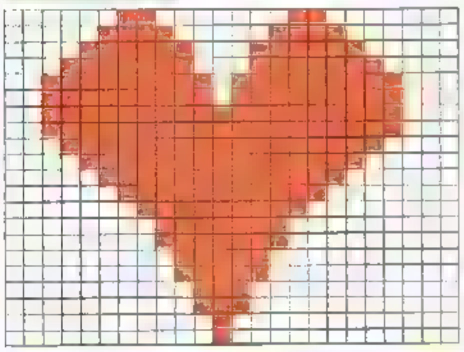


```
0,0,0,0,0,0,6
205,128,6,205,128,7,243
128,7,241,128,1,254,0
1,250,0,3,243,0,1
250,0,1,250,0,1,250
0,1,250,0,1,250,0
1,250,0,3,253,0,7
241,128,7,251,128,3,255
0,7,249,128,15,255,192
```

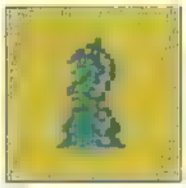


HEART

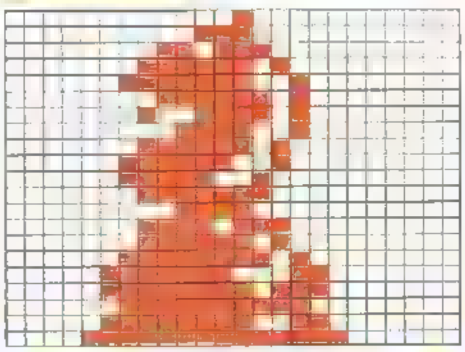
```
3,1,128,7,131,192,15
199,224,31,199,240,63,239
248,63,239,248,63,255,248
63,255,248,31,255,240,15
255,224,7,255,192,3,255
128,1,255,0,0,254,0
0,254,0,0,124,0,0
124,0,0,56,0,0,56
0,0,16,0,0,16,0
```



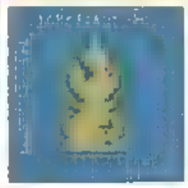
KNIGHT



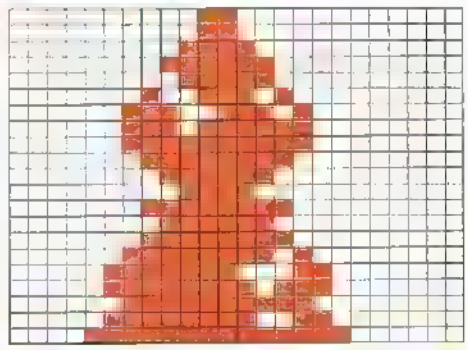
```
0,8,0,0,56,0,0
220,0,1,250,0,3,249
0,0,253,0,1,61,0
0,121,0,1,250,0,3
242,0,3,228,0,1,252
0,0,120,0,1,238,0
1,250,0,3,253,0,7
241,128,7,251,128,3,255
0,7,249,128,15,255,192
```



BISHOP

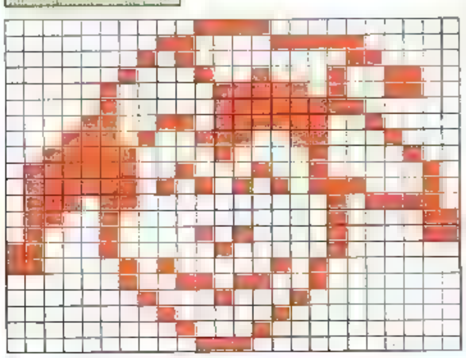


```
0,48,0,0,48,0,0
120,0,0,252,0,0,124
0,1,58,0,3,157,0
3,191,0,3,255,0,1
254,0,0,252,0,0,120
0,1,254,0,0,252,0
1,254,0,3,253,0,7
241,128,7,251,128,3,255
0,7,249,128,15,255,192
```



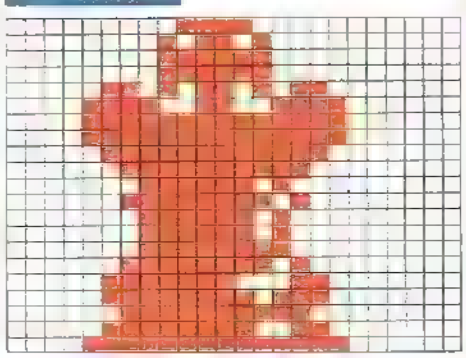
JOKER

```
0,60,0,0,195,224,1
64,48,2,36,76,4,31
76,4,31,224,12,223,144
29,57,136,62,16,132,126
68,130,126,170,226,118,0
94,100,0,67,68,40,67
196,16,64,194,130,128,2
108,128,1,17,0,0,130
0,0,68,0,0,56,0
```

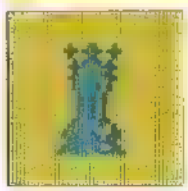


KING

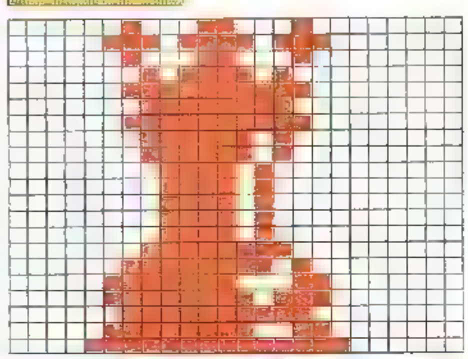
```
0,120,0,0,180,0,0
252,0,0,252,0,6,181
128,15,51,192,15,255,192
15,255,192,7,255,128,3
255,0,1,250,0,3,253
0,1,250,0,1,250,0
1,250,0,3,253,0,7
241,128,7,251,128,3,255
0,7,249,128,15,255,192
```



QUEEN



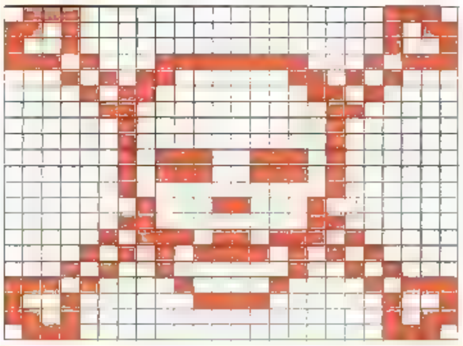
```
2,49,0,7,123,128,2
49,0,1,122,0,3,255
0,1,254,0,3,255,0
1,250,0,1,250,0,0
244,0,0,244,0,0,244
0,0,244,0,0,244,0
1,254,0,3,253,0,7
241,128,7,251,128,3,255
0,7,249,128,15,255,192
```



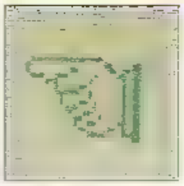
GAMES SYMBOLS

SKULL AND CROSSBONES

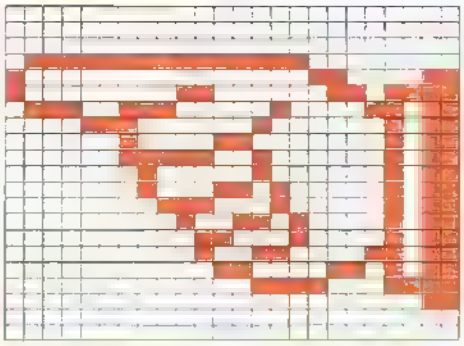
```
112,0,14,208,0,11,144
0,9,232,255,23,21,129
168,11,0,208,6,0,96
2,0,64,2,0,64,2
231,64,2,231,64,2,0
64,1,24,128,3,0,192
5,219,160,10,189,80,20
129,40,232,66,23,144,60
9,208,0,11,112,0,14
```



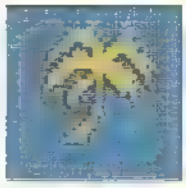
POINTING HAND



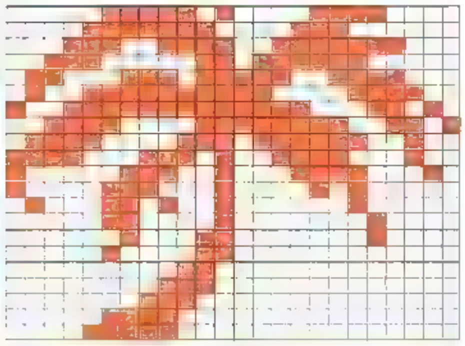
```
0,0,0,0,0,0,0
0,0,127,255,0,128,0
195,128,96,111,115,130,27
14,4,11,2,24,11,3
228,11,1,4,11,1,26
11,3,226,11,3,77,11
3,49,11,6,38,27,0
24,107,0,7,143,0,0
3,0,0,0,0,0,0
```



PALM TREE

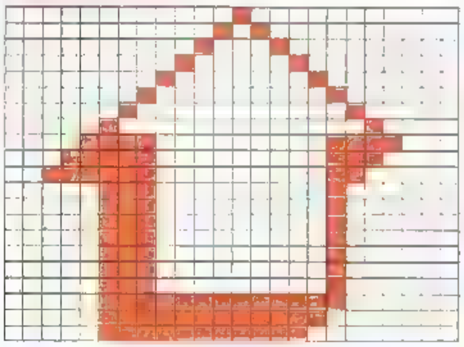


```
7,208,240,15,225,192,28
243,248,56,55,224,99,190
120,79,252,60,159,255,28
60,63,205,120,119,228,113
211,196,195,145,226,135,16
160,134,144,32,70,16,16
2,48,15,4,48,0,0
96,0,0,224,0,1,192
0,7,129,0,15,128,0
```



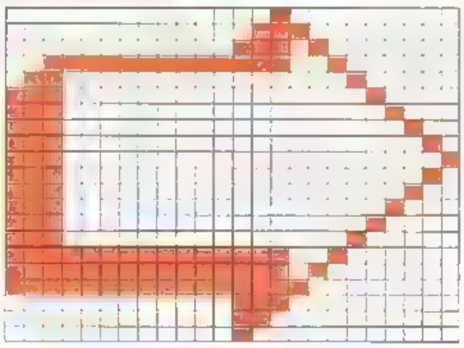
ARROW

```
0,8,0,0,20,0,0
34,0,0,65,0,0,128
128,1,0,64,2,0,32
4,0,16,15,0,120,31
0,112,53,0,96,7,0
64,7,0,64,7,0,64
7,0,64,7,0,64,7
0,64,7,0,64,7,255
192,7,255,128,7,255,0
```

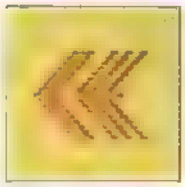


ARROW

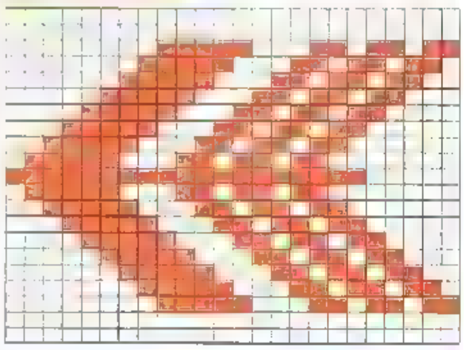
```
0,2,0,0,0,0,0
14,128,63,254,64,96,0
32,224,0,16,224,0,8
224,0,4,224,0,2,224
0,1,224,0,2,224,0
4,224,0,8,224,0,16
224,0,32,255,224,64,255
224,128,255,255,0,0,14
0,0,12,0,0,8,0
```



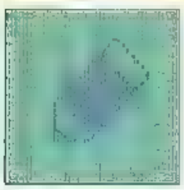
ARROW



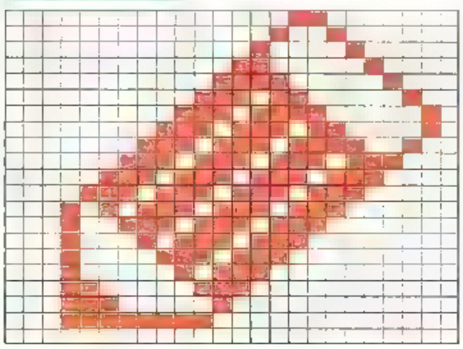
```
0,0,0,0,0,0,0
248,219,1,241,182,3,227
108,7,158,216,15,141,176
31,27,96,62,54,192,124
109,128,255,255,224,124,109
128,62,54,192,31,27,96
15,141,176,7,198,216,3
227,108,1,241,182,0,248
219,0,0,0,0,0,0
```



PENCIL

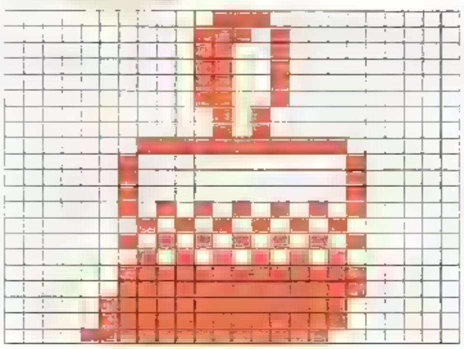


```
0,1,128,0,2,64,0
4,32,0,12,16,0,30
8,0,59,4,0,119,130
3,238,194,1,221,228,3
187,184,7,119,112,14,238
224,21,221,192,19,187,128
17,119,0,16,238,0,16
92,0,24,56,0,28,16
0,31,224,0,0,0,0
```

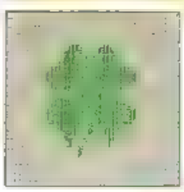


PAINTBRUSH

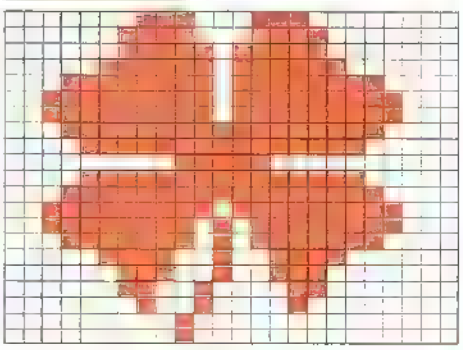
```
0,28,0,0,58,0,0
50,0,0,50,0,0,50
0,0,50,0,0,20,0
0,20,0,1,255,192,2
0,32,2,0,32,2,0
32,2,170,160,2,85,64
2,170,160,2,85,64,3
255,224,3,255,224,7,255
192,7,255,192,15,255,128
```



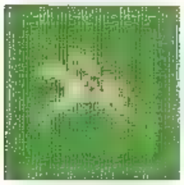
SHAMROCK



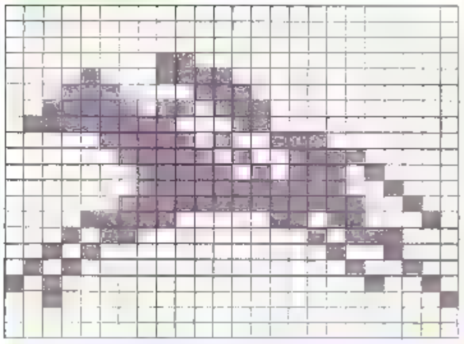
```
1,199,0,3,199,128,3
239,128,7,239,192,31,239
240,63,239,248,63,239,248
31,255,240,15,255,224,0
124,0,15,255,224,31,255
240,63,239,248,63,215,248
31,215,240,7,147,192,3
147,128,3,33,128,1,33
0,0,64,0,0,64,0
```



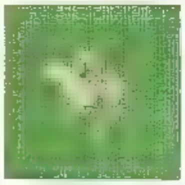
HORSE AND JOCKEY



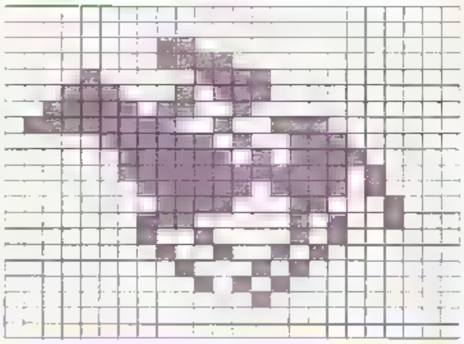
```
0,0,0,0,0,0,0
0,0,0,192,0,6,240
0,28,56,0,62,220,0
127,240,0,7,211,128,3
211,224,1,245,208,1,255
200,3,255,228,15,131,98
22,0,184,40,0,72,80
0,36,160,0,18,32,0
1,0,0,0,0,0,0
```



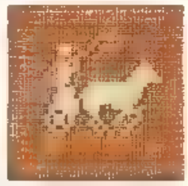
HORSE AND JOCKEY



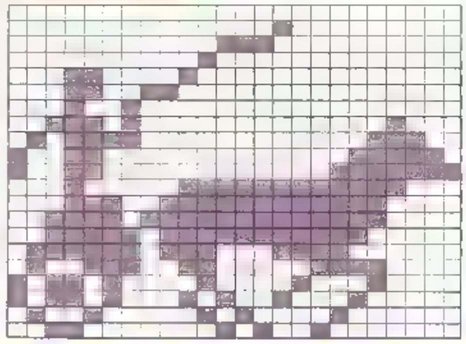
```
0,0,0,0,0,0,0
192,0,0,240,0,6,60
0,28,32,0,62,220,0
127,247,128,7,233,192,3
245,224,3,255,208,1,251
208,0,241,136,1,193,72
1,35,54,0,145,128,0
59,0,0,42,0,0,4
0,0,0,0,0,0,0
```



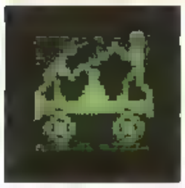
CHARIOT



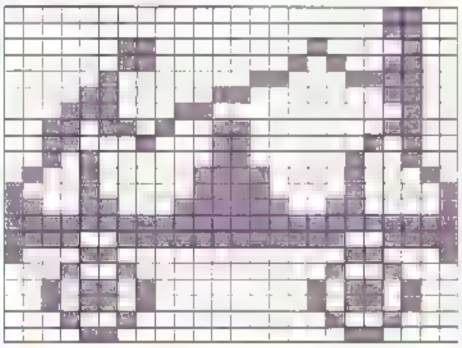
```
0,0,0,0,2,0,0
28,0,0,32,0,24,64
0,25,128,0,16,0,0
58,0,8,94,0,28,152
0,62,152,0,127,16,127
248,28,255,240,61,255,240
62,255,224,126,227,192,78
96,240,180,33,16,180,82
32,72,40,64,48,20,128
```



TROLLEY

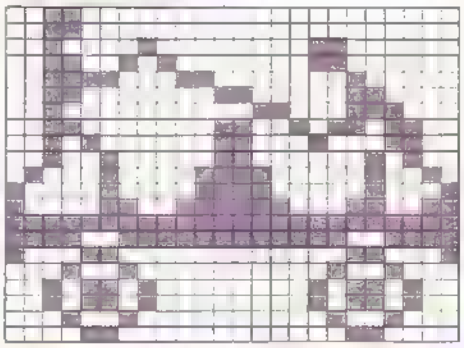


```
0,0,12,0,0,12,3
0,132,3,1,76,4,6
60,12,24,22,28,96,4
54,152,12,49,24,20,40
24,36,72,60,34,136,60
33,204,126,97,255,255,255
243,255,207,140,0,49,18
0,72,45,0,180,45,0
180,18,0,72,12,0,48
```

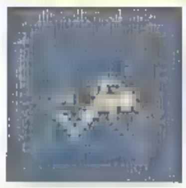


TROLLEY

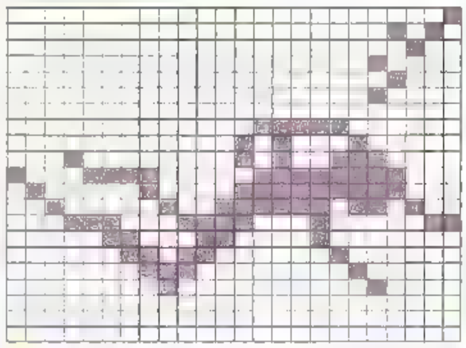
```
48,0,0,48,0,0,33
0,192,50,128,192,60,96
32,48,24,48,32,6,56
48,25,100,40,24,140,36
24,20,60,60,16,132,60
17,134,126,51,255,255,255
243,255,207,140,0,49,18
0,72,45,0,180,45,0
160,18,0,72,12,0,48
```



FROGMAN

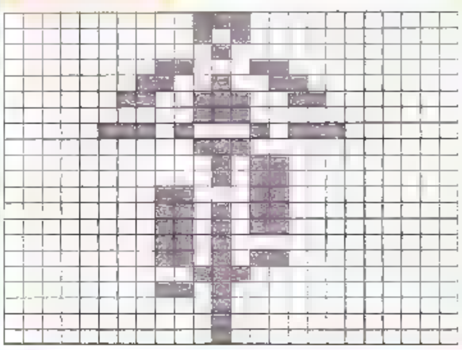


```
0,0,1,0,0,10,0
0,4,0,0,16,0,0
8,0,0,16,0,0,0
0,7,192,0,10,32,16
10,112,143,7,248,65,15
248,32,159,180,28,120,131
6,48,128,3,96,64,2
192,32,0,128,16,0,0
0,0,0,0,0,0,0
```

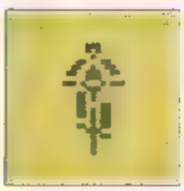


BMX RIDER

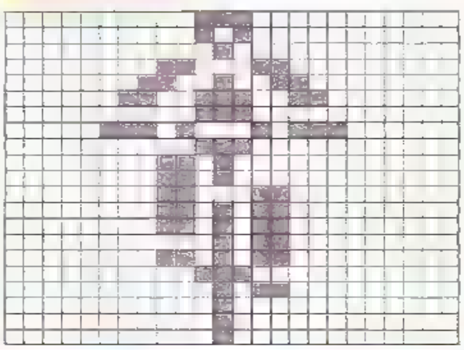
```
0,56,0,0,40,0,0
16,0,0,198,0,1,147
0,3,57,128,0,56,0
7,69,192,0,56,0,0
22,0,0,22,0,0,198
0,0,214,0,0,214,0
0,208,0,0,214,0,0
56,0,0,208,0,0,16
0,0,16,0,0,16,0
```



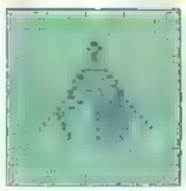
BMX RIDER



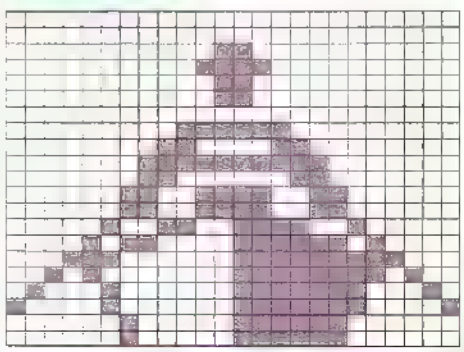
```
0,56,0,0,40,0,0
16,0,0,198,0,1,147
0,3,57,128,0,56,0
7,69,192,0,56,5,0
208,0,0,208,0,0,198
0,0,214,0,0,214,0
0,22,0,0,214,0,0
56,0,0,22,0,0,16
0,0,16,0,0,16,0
```



MAN IN BOAT



```
0,0,0,0,0,0,0
24,0,0,60,0,0,24
0,0,24,0,0,0,0
0,126,0,0,195,0,1
255,128,1,129,128,3,60
192,3,60,192,4,207,32
11,15,208,28,15,248,40
15,244,68,15,226,132,15
225,2,15,192,2,15,192
```



ROUTINES CHECKLIST

The table shown below gives a summary of all the machine-code routines used in this book. This table does not explain every detail of using each routine; it is intended only as an aid when using the routines in your programs. If you have not used a routine before, it is recommended that you read the introduction to the routine on the appropriate page of the book before using it in your program.

page	routine	parameters			co-ordinates
	sprite buffer				
13	24 x 21 sprite editor	Fna ()–e()			character
15	sprite print	FNf (x,y,n)	x,y	print position	–
16	master sprite	– – –			
17	sprite handling	FNg(x,y,d,l,s,c,n)	x,y	start co-ordinates	pixel
			d	direction	–
			l	distance to be moved	pixel x 3
			s	switch	–
			c	collision flag	–
			n	sprite number	–
21	keyboard-controlled sprite	FNh (s,x,y,c,n)	s	switch	–
			x,y	start co-ordinates	pixel
			c	collision flag	–
			n	sprite number	–
	interrupt vector table				
22	double horizontal sprite	FNi (x,y,d,l,s,c,n)	x,y	start co-ordinates	pixel
			d	direction	–
			l	distance to be moved	pixel x 3
			s	switch	–
			c	collision flag	–
			n	sprite number	–
23	double vertical sprite	FNj (x,y,d,l,s,c,n)	x,y	start co-ordinates	pixel
			d	direction	–
			l	distance to be moved	pixel x 3
			s	switch	–
			c	collision flag	–
			n	sprite number	–
25	sprite animation	FNk (x,y,d,l,s,f,c,v,n)	x,y	start co-ordinates	pixel
			d	direction	–
			l	distance to be moved	pixel x 3
			s	switch	–
			f	number of frames	–
			c	collision flag	–
			v	animation speed	–
			n	sprite number	–
29	horizontal scroll	FNl (l,d)	l	length of scroll	pixel
			d	direction	–
29	vertical scroll	FNm(l,d)	l	length of scroll	pixel
			d	direction	–
31	window	FNn (x,y,l,n,d,r)	x,y	start co-ordinates	character
			l	width of window	character
			n	sprite number	–
			d	direction	–
			r	repeat flag	–
33	interrupt-driven window	FNo (s,x,y,l,n,d,r)	s	switch	–
			x,y	start co-ordinates	character
			l	width of window	character
			n	sprite number	–
			d	direction	–
			r	repeat flag	–

Before using a routine, you must first define it in your program using DEF FN followed by the correct number of parameters. Parameters passed to machine-code routines must always be whole numbers; if a parameter value is calculated by your program, then put an INT statement in front of it to ensure a whole-number value is passed to the routine.

ranges	bytes	address	check
	700	54600	
	355	54200	190
0-28 and 0-20	75	54100	60
1-10	365	53700	83
0-231 and 0-154 0-3 0–51 (vertical) 0-77 (horizontal) 0-1 0-1 1-10	170	53500	66
0-1 0-231 and 0-154 0-1 1-10	250	53100	26
	256	52736	
0-231 and 0-154 0-3 0-51 (vertical) 0-77 (horizontal) 0-1 0-1 1-10	235	52400	53
0-231 and 0-154 0-3 0-51 (vertical) 0-77 (horizontal) 0-1 0-1 1-10	230	52100	20
0-231 and 0-154 0-3 0-51 (vertical) 0-77 (horizontal) 0-1 1-10 0-1 1-255 1-10	275	51700	63
0-255 0-1	190	51500	61
0-175 0-1	215	50900	47
0-31 and 0-21 0-31 1-10 0-1 0-1	290	49600	43
0-1 0-31 and 0-21 0-31 1-10 0-1 0-1	315	49200	154

MEMORY MAP

This chart shows how the Spectrum memory is organized when all the routines are present in memory. RAMTOP can be set to 49000 using a CLEAR command.

routine	title	address
	lowest book 3 routine	55500
	sprite buffer (700 bytes)	54600
FNa – FNe	24 x 21 sprite editor	54200
FNf	sprite print	54100
	master sprite	53700
FNg	sprite-handling	53500
FNh	keyboard-controlled sprite	53100
	interrupt vector table	52736
FNi	double horizontal sprite	52400
FNj	double vertical sprite	52100
FNk	sprite animation	51700
FNl	horizontal scroll	51500
FNm	vertical scroll	50900
FNn	window	49600
FNo	interrupt-driven window	49200
	RAMTOP (after CLEAR 49000)	49000

INDEX

Main entries are given in **bold** type

Acknowledgments

A number of people helped and encouraged me with this book. Thanks to Alan and Michael at Dorling Kindersley, to Jacqui Lyons for her representation and to Andy Werbinski for reluctant assistance. I am particularly grateful, as always, to my parents, and to Martine.

Piers Letcher
Spring 1985

DK

Screen Shot

The bestselling teach-yourself programming course now offers the first complete full-colour book on creating sprites on the ZX Spectrum.

Illustrated with over 300 screen-shot photographs, it contains programs for single and double sprites, for animation, and for setting overlaps and detecting collisions, and includes an easy-to-use sprite generator with which you can design and save your own sprites. In addition, there is a full-colour design directory containing over 200 original sprite designs complete with all the data needed to program them.

Together, Books Three and Four in this series form a complete, self-contained graphics system for Spectrum-owners.

All the programs in this book run on both 48K ZX Spectrum and ZX Spectrum+ machines.

“Far better than anything else reviewed on these pages . . . Outstandingly good”
BIG K

“As good as anything else that is available, and far better than most”
COMPUTING TODAY

“Excellent . . . As a series they could form the best 'basic introduction' to programming I've seen”
POPULAR COMPUTING WEEKLY

DORLING KINDERSLEY

ISBN 0-86318-104-X

£5.95

9 780863 181047

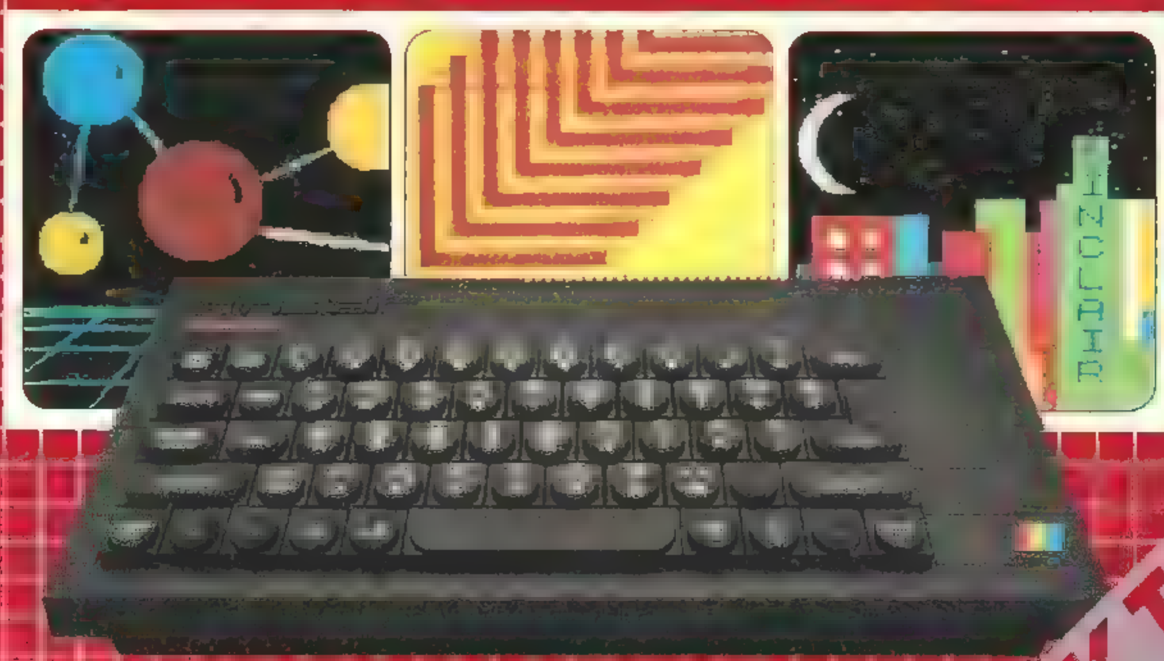
DK

ScreenShot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING ZX SPECTRUM AND ZX SPECTRUM+

GRAPHICS



PIERS LETCHER

BOOK THREE

A complete BASIC plus machine-code system for fast high-resolution graphics

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM ZX SPECTRUM +

GRAPHICS

THE DK SCREEN-SHOT PROGRAMMING SERIES

Books One and Two in the DK Screen-Shot Programming Series brought to home computer users a new and exciting way of learning how to program in BASIC. Following the success of this completely new concept in teach-yourself computing, the series now carries on to explore the speed and potential of machine-code graphics. Fully illustrated in the unique Screen-Shot style, the series continues to set new standards in the world of computer books.

BOOKS ABOUT THE ZX SPECTRUM+

This is Book Three in a series of guides to programming the ZX Spectrum+. It contains a complete BASIC-and-machine-code graphics language for the Spectrum+, and features its own graphics editor which enables you to use all these facilities directly from the keyboard. Together with its companion volumes, it builds up into a complete programming and graphics system.

ALSO AVAILABLE IN THE SERIES

Step-by-Step Programming for the **Commodore 64**

Step-by-Step Programming for the **BBC Micro**

Step-by-Step Programming for the **Acorn Electron**

Step-by-Step Programming for the **Apple IIe**

Step-by-Step Programming for the **Apple IIc**

PIERS LETCHER

After graduating with a degree in Computer Systems, Piers Letcher has worked in many areas of the computer industry, from programming and selling mainframes to designing and marketing educational software. He was Peripherals Editor of *Personal Computer News* until May 1984 and has since written a guide to peripherals and a number of other books for popular home micros.

BOOK THREE


TRUE VIDEO
EXTEND MODE

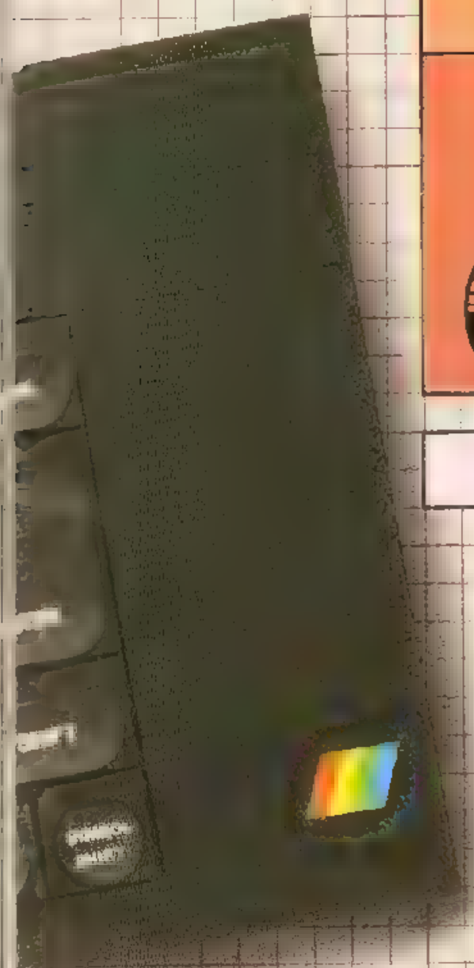
DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM
ZX SPECTRUM +
GRAPHICS

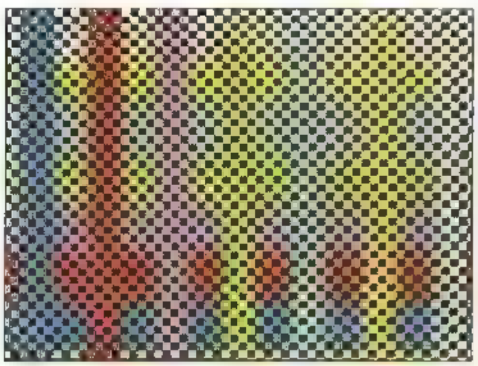


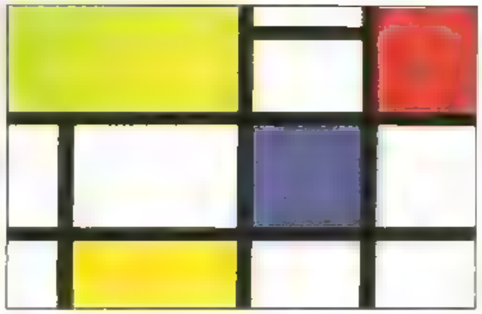
PIERS LETCHER

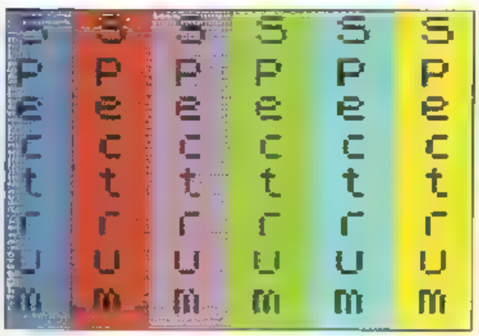
DORLING KINDERSLEY·LONDON

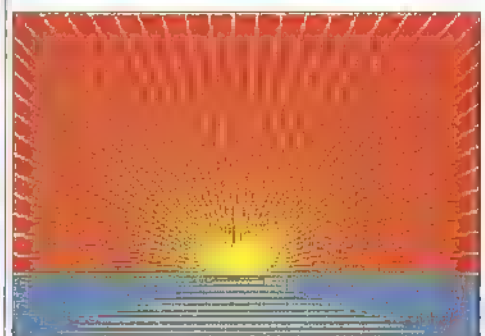
BOOK THREE

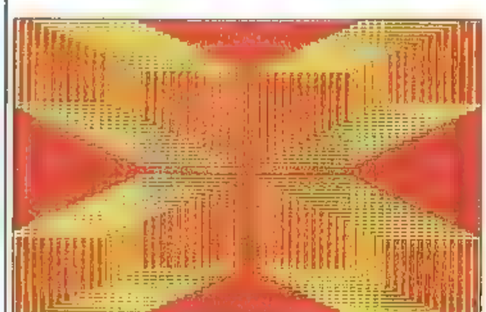
CONTENTS











The DK Screen-Shot Programming Series was conceived, edited and designed by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

Editor Michael Upshall
Designer Steve Wilson
Photographer Vincent Oliver
Series Editor David Burnie
Series Art Editor Peter Luff
Managing Editor Alan Buckingham

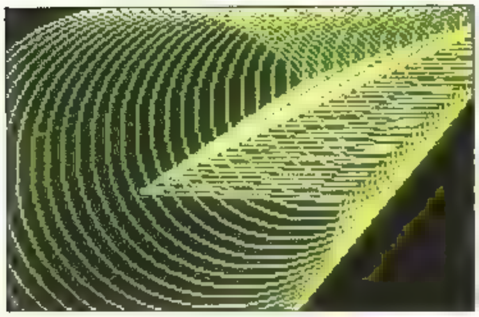
First published in Great Britain in 1985 by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.
Second impression 1985

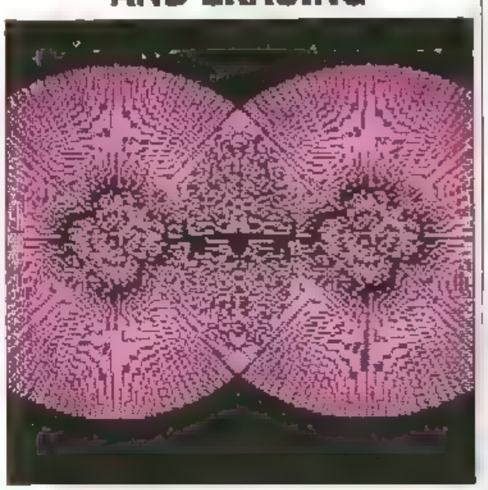
British Library Cataloguing in Publication Data

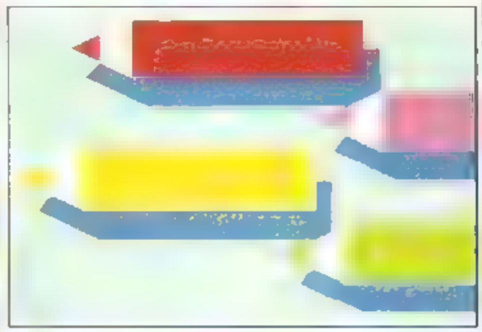
Letcher, Piers
Step-by-step programming ZX Spectrum and ZX Spectrum+ Graphics.
– (DK screen shot programming series) Bk. 3
1. Sinclair ZX Spectrum (Computer) – Programming
I. Title
001.64'2 QA76.8.S625

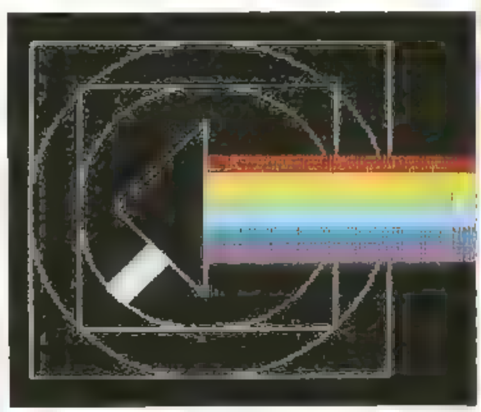
ISBN 0-86318-103-1

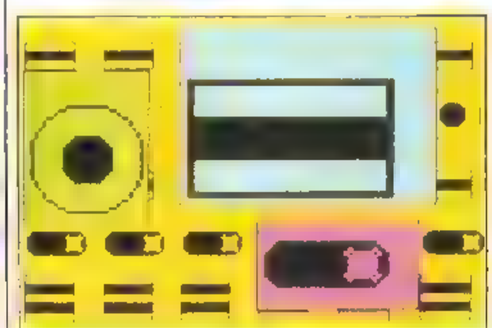
Typesetting by Gedset Limited, Cheltenham, England
Reproduction by Reprocolor Llovet S.A., Barcelona, Spain
Printed and bound in Italy by A. Mondadori, Verona











ABOUT THIS BOOK

The Sinclair Spectrum is one of the most popular microcomputers ever produced. One reason for its success has been its remarkable ability to produce graphic displays rivalling those produced by much larger computers designed only ten or fifteen years ago. However, graphics programming in BASIC under-utilizes the Spectrum. To produce the kind of displays seen in commercially available games, you need to use machine code as well as BASIC.

What is machine code?

The heart of the Spectrum, the Z80 central processor, cannot understand BASIC. A BASIC program must first be translated into a simpler language that the machine can understand (hence the term "machine code"). This code is in the form of binary 1s and 0s. Before the processor can execute a BASIC program line, all keywords and variables are first converted to machine-code instructions.

BASIC is an example of what is known as an "interpreted", as opposed to a "compiled", language — that is, it is executed by the central processor line by line rather than as a complete program. While an interpreted language is easier to use, it is also slower in execution. By writing programs in machine code, you can miss out the BASIC interpreter altogether. In addition, machine code allows you to utilize many features of your Spectrum which cannot be reached from BASIC, so that you can therefore achieve far more impressive results than would ever be possible from the simpler, but more restricted, BASIC. You can get an idea of how much faster machine code is by seeing the time taken for the programs in this book to run.

Disadvantages of machine code

Given all the advantages of machine code in both speed and flexibility, why not ignore BASIC and use machine code all the time? The answer is simply convenience. Using machine code is time-consuming, difficult and frustrating, and attempting to write your own code is only for the expert. When you see machine-code listings, they are usually in a "disassembled" form, that is, with some of the numbers translated into mnemonics such as LD for LOAD, and JP for JUMP. But a special disassembler program is required simply to give you a machine-code listing in this form, and these mnemonics are themselves far from simple. Using machine code even the simplest operations in BASIC, such as drawing a line on the screen, require many lines of machine code. In addition, machine code has no error-trapping routines such as those in BASIC. If a mistake is made when keying in a BASIC program, the program will not be lost (although the program may refuse to RUN at some point); in machine code, without error-trapping routines, a mistake will probably cause the Spectrum to crash, erasing both the program and its DATA.

The solution

This book combines the advantages of machine code with the convenience and simplicity of BASIC. This is done by giving the machine code in the form of ready-made and tested routines, which you can then use in your BASIC programs. The machine code is shown as DATA statements in BASIC, which means it isn't necessary for you to understand anything about machine code to be able to use the routines. The DATA is given in the form of decimal numbers, rather than in binary or hexadecimal (to base 16), so that the machine code is in the form most convenient for you to read and key in.

The machine-code routines

Here is an example of a machine code routine (the point-plot routine, FNf, from page 17):

ROUTINE LISTING

```
7350 LET b=61500: LET l=60: LET
z=0: RESTORE 7360
7351 FOR i=0 TO l-1: READ a
7352 POKE (b+i),a: LET z=z+a
7353 NEXT i
7354 LET z=INT (((z/l)-INT (z/l)
)*l)
7355 READ a: IF a<>z THEN PRINT
"??": STOP

7360 DATA 42,11,92,1,4
7361 DATA 0,9,86,14,8
7362 DATA 9,94,62,175,147
7363 DATA 216,95,167,31,55
7364 DATA 31,167,31,171,230
7365 DATA 248,171,103,122,7
7366 DATA 7,7,171,230,199
7367 DATA 171,7,7,111,122
7368 DATA 230,7,71,4,62
7369 DATA 254,15,16,253,6
7370 DATA 255,168,71,126,176
7371 DATA 119,201,0,0,0
7372 DATA 24,0,0,0,0
```

Each routine in the book is shown like this, in the form of a BASIC program. The machine code is contained as a series of DATA statements in lines 7360-7372. At the beginning of the routine in lines 7350-7355, there are a few lines of BASIC. This is a loader program; variable b tells the computer where in memory to begin loading the routine, and variable l the number of bytes in the routine. When the loader routine is RUN, this routine is placed in memory from address 61500 onwards, and has a length of 60 bytes.

As shown here, of course, the routine is simply a list of numbers, and has no visible meaning. These numbers are the ready tested and assembled machine code which has then been converted to a sequence of decimal numbers. Each number corresponds to a single instruction or item of DATA required by the routine;

hence, all the numbers have values between 0 and 255, the maximum value of a byte. All you need to know about the routine is what it does and what information it requires so that you can call it correctly from your BASIC program.

All the routines in the book are defined as functions. Each function is individually coded by the letters a to t; a complete list of functions is given on pages 60-61. Demonstration BASIC programs can be found on the same page as the routine, which give an indication of the kind of displays possible using the machine code.

How to use the routines

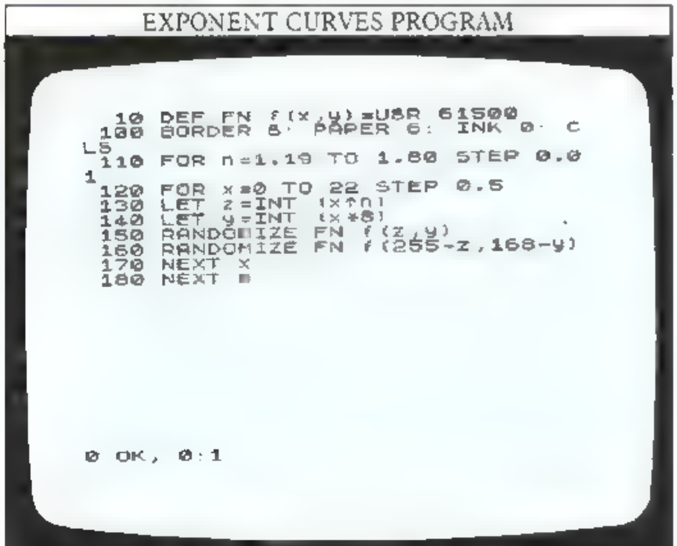
To use any program in this book, simply key in a machine-code routine together with a BASIC program which demonstrates its use. You will find full details of how to do this on pages 8-9. When you RUN the program, you will immediately begin to see the true power of your Spectrum.

As you progress through the book and the range of routines grows, the BASIC programs grow too by calling several routines to produce increasingly complex graphics. By keying in each routine just once, and then SAVEing it onto cassette or Microdrive, you will have a sophisticated graphics capability at your fingertips.

The programs in use

A typical program from this book (the exponent curves program on page 17) contains two details which will be unfamiliar to BASIC programmers who have not used machine code before:

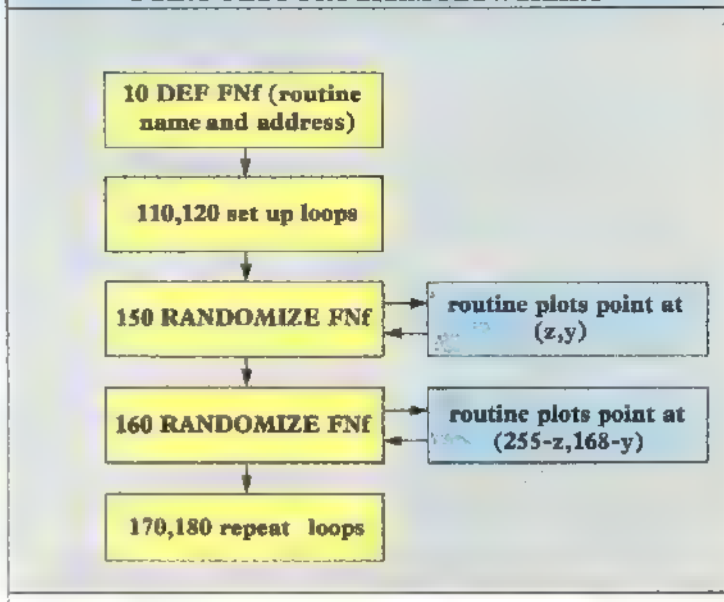
EXPONENT CURVES PROGRAM



```
  10 DEF FN f(x,y)=USR 61500
 100 BORDER 6: PAPER 6: INK 0: C
LS
 110 FOR n=1.19 TO 1.80 STEP 0.0
1
 120 FOR x=0 TO 22 STEP 0.5
 130 LET z=INT (x↑n)
 140 LET y=INT (x*8)
 150 RANDOMIZE FN f(z,y)
 160 RANDOMIZE FN f(255-z,168-y)
 170 NEXT x
 180 NEXT n

0 OK, 0:1
```

First, you will see in line 10 a DEF FN statement, which is used to instruct the computer that a machine-code routine with two parameters (x and y) is located at address 61500 in memory. You will also notice two RANDOMIZE FN commands (lines 150 and 160). These are the calls to the point-plot routine, and the numbers in brackets which follow them are the parameter values to be passed to the machine-code routine (in this case, the co-ordinates of the point to be plotted). When RUNning, the program is carried out by the computer in this way:

POINT-PLOT PROGRAM FLOWCHART



On the left side of this diagram is the main BASIC program, and on the right you can see the machine-code routines, called twice using a RANDOMIZE FN statement. You will see from the diagram that the machine-code is used here very much as a subroutine would be used in BASIC, with variables passed to the routines each time they are called.

What the routines do

Much of this book gives you machine-code versions of the graphics commands you are familiar with in BASIC. You will find that the machine code is often many times faster, and offers you alternative ways of producing graphics which will often be preferable to the BASIC method.

In addition, several routines are included in this book which would simply not be possible in Sinclair BASIC, such as magnification and reduction, and filling in irregular shapes with the current INK colour.

The graphics editor

To make the machine-code routines in this book even easier to use, all the routines contained on pages 10 to 41 have been combined in a single program to form a complete package of routines, which you can use as a graphics editor (pages 42 to 51). No knowledge of BASIC is required to use the graphics editor; even someone with no knowledge of programming, and who has never used a Spectrum before will quickly learn how to produce sophisticated displays using this graphics editor.

USING THE MACHINE CODE

The machine-code routines in this book can easily be incorporated into your BASIC programs without you having to understand the intricacies of how they work. Simply choose a program from this book, and follow the four steps given here.

1: CLEAR memory

As soon as you switch on the Spectrum, type CLEAR 55500. This command resets RAMTOP, the top of the area in memory free for BASIC programs, and ensures that BASIC programs cannot overlap with the machine code (stored in memory from 55500 upwards). Now you can safely use NEW to delete BASIC programs without deleting any of the machine code in memory.

Remember to use CLEAR before loading machine code, since this command erases whatever is in memory above the specified address.

2: Load the machine code

Now type in whatever machine-code routines are required by the BASIC program. After keying in the routine, RUN the short BASIC program which accompanies it: this loads the code into memory. If you keyed in the DATA correctly, you will see an "OK" message on the screen; if not, you will see a couple of question marks. In this case, look again at what you have typed in to trace the mistake.

3: SAVE the routine

When you are sure you have keyed in the routine correctly, SAVE it onto cassette or Microdrive. Always SAVE machine code before using it, to minimize the risk of losing everything you have keyed in. When BASIC errors occur, an error message is usually produced but the program is not lost. Machine-code routines, however, do not generally have error-trapping facilities, and a fault in the code will as often as not cause the Spectrum to crash — deleting everything in memory.

The machine code can be SAVEd in two ways: either in the form of DATA statements like any other BASIC

EXPLANATION OF A MACHINE-CODE BOX

FNi

TRIANGLE DRAW ROUTINE

Start address 60300 **Length** 80 bytes
Other routines called Line draw routine (FNg).
What it does Draws a triangle given the pixel co-ordinates of three points.

Using the routine The routine uses absolute co-ordinates. Specifying off-screen co-ordinates produces an error message; values more than 255 pixels off the screen will probably cause the Spectrum to crash. Colours are set by the current screen INK attributes.

ROUTINE PARAMETERS

DEF FNi(x,y,p,q,r,s)

x,y	specify first corner of triangle (x<256,y<176)
p,q	specify second corner of triangle (p<256,q<176)
r,s	specify third corner of triangle (r<256,s<176)

ROUTINE LISTING

```
7600 LET b=60300  LET l=75  LET
z=0  RESTORE 7610
7601 FOR i=0 TO l-1. READ a
7602 POKE (b+i),a. LET z=z+a
7603 NEXT i
7604 LET z=INT (((z/l)-INT (z/l)
)*l)
7605 READ a: IF a<>z THEN PRINT
"??": STOP

7610 DATA 42,11,92,1,4
7611 DATA 0,9,86,14,8
7612 DATA 9,94,237,83,208
7613 DATA 235,9,86,9,94
7614 DATA 237,83,210,235,9
```

Routine title (a single letter)

Name of routine

Address in memory at which routine is located

Number of bytes in memory taken up by routine

Other machine-code routines which must be present in memory for this routine to work

Purpose of routine

Points to note when using the routine

List of parameters used by the routine, and letters used to describe these parameters

What the parameters do

Maximum and minimum values of parameter to ensure the routine does not plot off-screen points

BASIC loading routine for machine-code DATA

Number of bytes for machine-code (without check digit)

Start address for POKEing DATA

Calculates check digit z

POKEs byte value a into location (b+i)

READs next DATA item, the routine check digit; if this is not the same as z, two question marks are PRINTed to show a mistake has been made

Start of machine-code DATA

listing, or, after you have loaded it into memory, as a block of code. To save machine code, type:

SAVE "routine name" CODE start address, length in bytes

The start address and length are given at the top of each machine-code box. The diagram on the facing page shows how this information is displayed.

4: LOAD a BASIC program

With the machine-code routine in memory, you can now use it in a BASIC program. DEF FN statements are used to tell the Spectrum the whereabouts of the routine in memory, and what information the routine requires.

Using functions

A machine-code routine can be called simply by specifying its start location, like this:

```
10 RANDOMIZE USR 63000
```

A line like this in a BASIC program, however, is not very informative. It tells you neither what the routine does, nor how many parameters the routine may require when called. This information could be POKEd into the appropriate memory locations — but the consequences of a mistake would be disastrous. Much more reliable is to pass information to the routines using a BASIC function. Functions on the Spectrum are identified by a single letter, and are followed by parameters in brackets. When you define the name and location of the function in your program, you must also specify the parameters, if any, which are to be passed to the routine. For example, the screen clear routine, FNa, requires four parameters:

```
10 DEF FN a(x,y,h,v) = USR 63000
```

Which letters are used after DEF FN is not important; their function is only to tell the computer the number of parameters which will follow the routine call in a BASIC program.

A machine-code function can be called from BASIC in two main ways, both of which require you to combine the keywords FN or USR with a BASIC keyword.

The method used generally in this book is with the keyword RANDOMIZE. Thus,

```
RANDOMIZE FN a(10,10,10,10)
```

would clear a rectangular area of 10x10 characters with the top corner at point 10,10. Note that using RANDOMIZE also resets the random number generator with a new seed; this may cause problems if you are also using a random function in your program.

The second word you can use to call machine code is RESTORE. However, RESTORE also resets the pointer to DATA statements when you use it — which is of course the purpose of the RESTORE statement. If you opt to use RESTORE instead of RANDOMIZE then be especially careful if there are any READ or DATA statements in your program.

QUESTIONS AND ANSWERS

What if I make a mistake in keying in?
Don't panic! Nobody keys in long lists of numbers without making any mistakes. A check routine is included with each machine-code routine to warn you if you made any mistakes in keying in the DATA. This routine compares the DATA you have entered with a check number, which is placed by itself on the last DATA line of each routine.

After the loading program has POKEd the DATA numbers into memory, it looks to see if the check digit is the same as the one currently calculated. If the two numbers are different, the program prints two question marks to show an error has been made. If this happens look through the numbers you have typed in to find the mistake. Having corrected the error, you may still find that the routine fails to load correctly; look to see if you have made more than one error.

Can I start anywhere in the book?
Yes, you can start on any page, but obviously when you key in a program it will not RUN unless the machine-code routine it calls is present in memory. Check before you begin if the program you want to RUN calls any machine-code routines you haven't already keyed in.

Can I use more than one machine-code routine in my programs?
Yes — you can use any combination of routines from this book together. The complete graphics editor program (pages 42-51) provides a convenient way of using machine-code routines together in a single program.

Can I adapt the BASIC programs?
Yes. You can edit the BASIC programs in any way you want to produce different displays, and you will find suggestions for variations throughout the book. One suggestion, though, if you are going to experiment with unusual or off-screen values for the machine-code parameters, is to SAVE what you have keyed in before experimenting. This will prevent you from losing hours of work at the keyboard!

Can I adapt the machine-code routines?
Yes, but at your own risk! Without a good understanding of machine code, it is highly unlikely that you will be able to alter any of the routines successfully. Much more probable is that the Spectrum would crash, with the result that both program and code are wiped from memory.

SCREEN COLOURS 1

The Spectrum CLS command is used to wipe off any ink from the screen, leaving the PAPER and INK attributes for the screen unchanged.

However, there are often occasions when you may want to clear a portion of the screen without disturbing the rest of the display. The partial screen clear routine, FNa, enables you to do this. It clears any rectangular portion of the screen, leaving the PAPER and INK attributes at their current setting. It is used in a program by first defining the function (as in line 10 of the program below), and then calling it with a RANDOMIZE FNa statement (line 160).

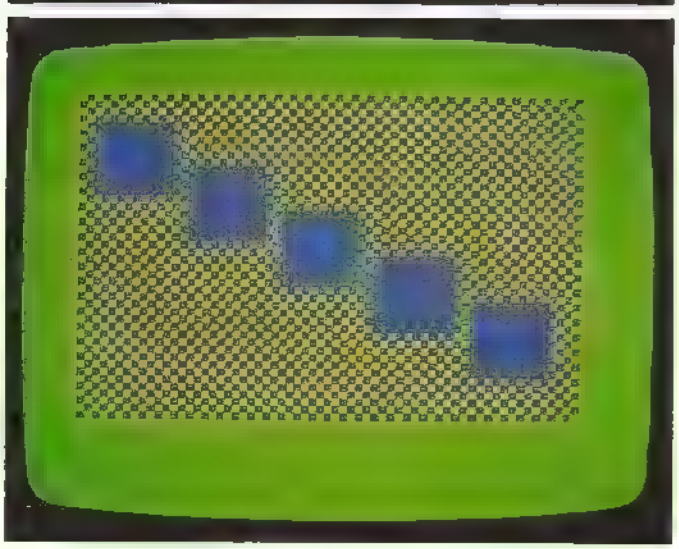
Colours on the Spectrum

The Spectrum screen colours are set by the familiar INK and PAPER commands. However, although each character square on the Spectrum can have one INK and one PAPER colour, a command such as INK 2 affects the whole screen, even though you may only want the command to affect a small area. How can this be done? It is possible to change the INK attribute of a single character square on the screen by PRINTing spaces in graphics mode and using OVER, or by finding the relevant memory location of the INK attribute, and POKEing the new value, but these would be very slow methods of changing the INK colour on more than a few squares.

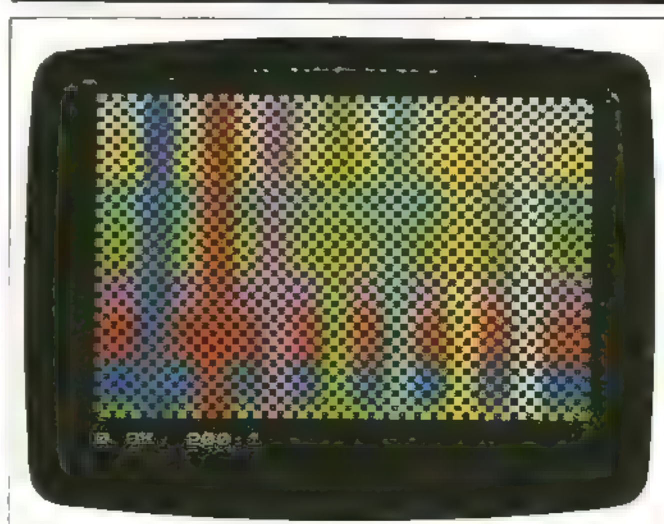
Changing colours with machine code

The window ink routine allows you to change the INK colour of any rectangular area of the screen. Whatever you draw on this area of the screen will now be in the INK colour specified by the routine, while outside this area, colours remain in the current Spectrum INK colour. The routine also allows you to set the BRIGHT and FLASH attributes of the area you are specifying.

PARTIAL SCREEN CLEAR PROGRAM



WINDOW INK PROGRAM



FNa

PARTIAL SCREEN CLEAR ROUTINE

Start address 63000 **Length** 100 bytes
What it does Similar to BASIC CLS command, but clears only a specified rectangular portion of the screen.

Using the routine Parameter values in this routine represent character positions rather than pixel positions. As with all the machine-code routines in this book, parameter values must be whole numbers.
If the result of adding x and h is greater than 31, or if the sum of y and v is greater than 23, the area to be cleared will run off the screen, and the routine may crash as a result.

ROUTINE PARAMETERS	
DEF FNa(x,y,h,v)	
x,y	specify top left-hand corner of area to be cleared (x<32, y<24)
h,v	horizontal and vertical size (x+h<32, y+v<24)

ROUTINE LISTING

```
7000 LET b=63000: LET l=95: LET
z=0
7001 FOR i=0 TO l-1: READ a
7002 POKE (b+i),a: LET z=z+a
7003 NEXT i
7004 LET z=INT (((z/l)-INT (z/l)
)*l)
7005 READ a: IF a<>z THEN PRINT
"??": STOP

7010 DATA 42,11,92,1,4
7011 DATA 0,9,86,1,8
7012 DATA 0,9,94,237,83
7013 DATA 116,246,9,86,9
7014 DATA 94,237,83,118,246
7015 DATA 237,91,116,246,123
7016 DATA 254,23,240,237,83
7017 DATA 116,246,123,230,24
7018 DATA 246,64,103,123,230
7019 DATA 7,183,31,31,31

7020 DATA 31,130,111,58,118
7021 DATA 246,71,197,229,6
7022 DATA 8,197,229,58,119
7023 DATA 246,71,175,119,35
7024 DATA 16,252,225,193,36
7025 DATA 16,240,225,193,62
7026 DATA 32,133,111,48,4
7027 DATA 62,8,132,103,16
7028 DATA 222,201,0,0,0
7029 DATA 82,0,0,0,0
```

While the standard Spectrum screen graphics area is 22 characters deep, with two lines reserved below this for text (lines 23 and 24), both the routines on this page can be used anywhere within the whole screen area, 32 characters wide by 24 deep.

The two programs on the facing page demonstrate the machine-code routines in action. Both programs begin by PRINTing a graphics character over the whole screen (lines 110-130). The partial screen clear program then clears five rectangular areas on the screen. The result is that the INK is deleted in these areas, while the paper colour remains unchanged. The window ink program calls the ink routine in two loops eight times across and down the screen, producing a grid effect.

FNb

WINDOW INK ROUTINE

Start address 62800 **Length** 135 bytes
What it does Sets the INK colour of any specified part of the screen.

Using the routine Be careful that you do not try to set the ink colours of points off the screen. Since parameters h and v are added to x and y respectively, this means that x+h should not be greater than 31, and y+v should not exceed 23. If they do, you may crash the Spectrum and lose the program you were working on.
If, when using the routine, it appears that nothing has happened, then either you have set the INK colour to what it was already, or the area you have altered contained no INK attributes. Try the routine again after printing something in the specified area.
Note that the routine can set the ink colour over the whole Spectrum 32x24 character screen, not just over the normal 32x22 graphics area.

ROUTINE PARAMETERS	
DEF FNb(x,y,h,v,c,b,f)	
x,y	specify top left-hand corner of box area (x<32, y<24)
h,v	specify horizontal and vertical sizes of area (x+h<32, y+v<24)
c	specifies ink colour (0<=c<=7)
b	specifies bright (1=bright, 0=off)
f	specifies flash (1=flash, 0=off)

ROUTINE LISTING

```
7050 LET b=62800: LET l=130: LET
 z=0: RESTORE 7060
7051 FOR i=0 TO l-1: READ a
7052 POKE (b+i),a: LET z=z+a
7053 NEXT i
7054 LET z=INT (((z/l)-INT (z/l)
)*l)
7055 READ a: IF a<>z THEN PRINT
"??": STOP

7060 DATA 42,11,92,1,4
7061 DATA 0,9,86,1,8
7062 DATA 0,9,94,237,83
7063 DATA 210,245,9,86,9
7064 DATA 94,237,83,208,245
7065 DATA 9,126,230,7,50
7066 DATA 207,245,9,126,230
7067 DATA 1,40,8,58,207
7068 DATA 245,246,64,50,207
7069 DATA 245,9,126,230,1

7070 DATA 40,8,58,207,245
7071 DATA 246,128,50,207,245
7072 DATA 237,91,210,245,58
7073 DATA 208,245,254,0,200
7074 DATA 237,83,210,245,123
7075 DATA 230,24,203,63,203
7076 DATA 63,203,63,246,88
7077 DATA 103,123,230,7,183
7078 DATA 31,31,31,31,130
7079 DATA 111,58,208,245,71

7080 DATA 197,229,58,209,245
7081 DATA 71,126,230,56,79
7082 DATA 58,207,245,177,119
7083 DATA 35,16,244,225,1
7084 DATA 32,0,9,193,16
7085 DATA 230,201,0,2,5
7086 DATA 53,0,0,0,0
```

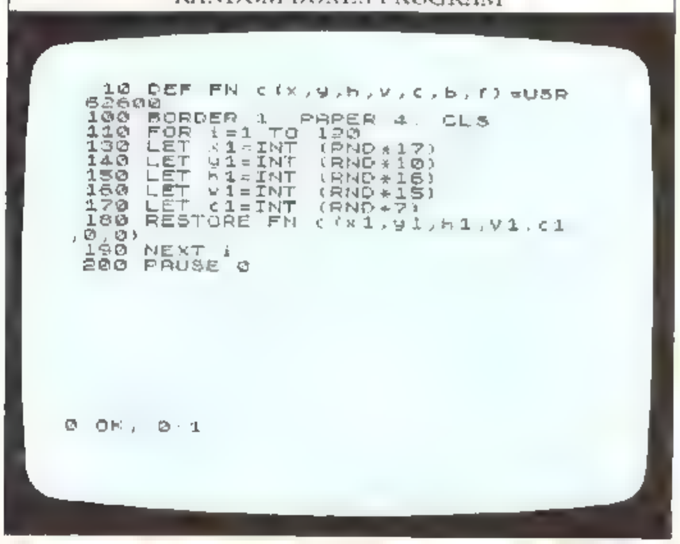
SCREEN COLOURS 2

The Spectrum PAPER command sets the background colour of the whole screen. The window paper routine on this page, FNc, allows you to set the paper colour of only a part of the screen, in the same way that you can use the window ink routine to change the INK colours on a part of the screen.

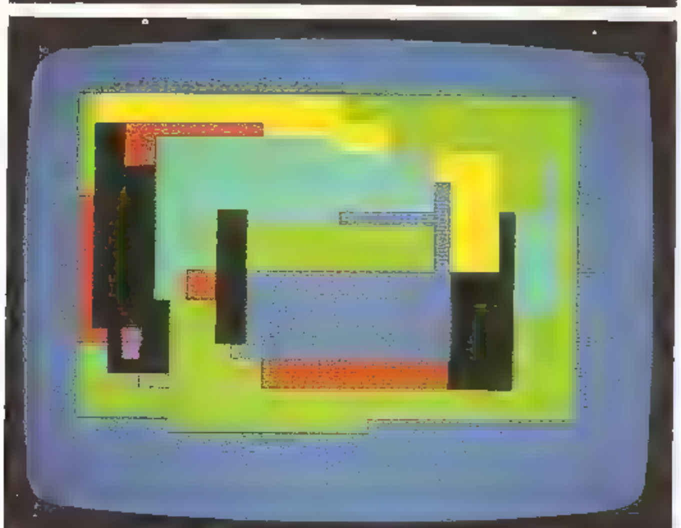
As for the ink routine, the paper routine requires you to specify the top left-hand co-ordinates and height and width of a rectangular area within which the colour is to be changed. Unlike the PAPER command in BASIC, you will see any colour change without having to clear the screen with CLS. Again, like the previous routine, you can use the routine to set the BRIGHT and FLASH attributes of the area. By calling the routine several times you can create a layered effect, with colours apparently superimposed on one another.

A layered effect forms the basis of the random boxes program on this page. Random values are chosen for the start co-ordinates (x1,y1) and horizontal and vertical increments (h1,v1) of the area, and a random colour value is chosen, before the routine is called, inside a loop. Note that the machine-code routine is called using RESTORE rather than RANDOMIZE. Using RANDOMIZE would reset the seed of the random number generator within the loop, so that the same random number sequence would begin again and again.

RANDOM BOXES PROGRAM



```
 10 DEF FN c(x,y,h,v,c,b,f)=USR
62600
100 BORDER 1: PAPER 4: CLS
110 FOR i=1 TO 120
130 LET x1=INT (RND*17)
140 LET y1=INT (RND*10)
150 LET h1=INT (RND*16)
160 LET v1=INT (RND*15)
170 LET c1=INT (RND*7)
180 RESTORE FN c(x1,y1,h1,v1,c1
,0,0)
190 NEXT i
200 PAUSE 0

0 OK, 0:1
```

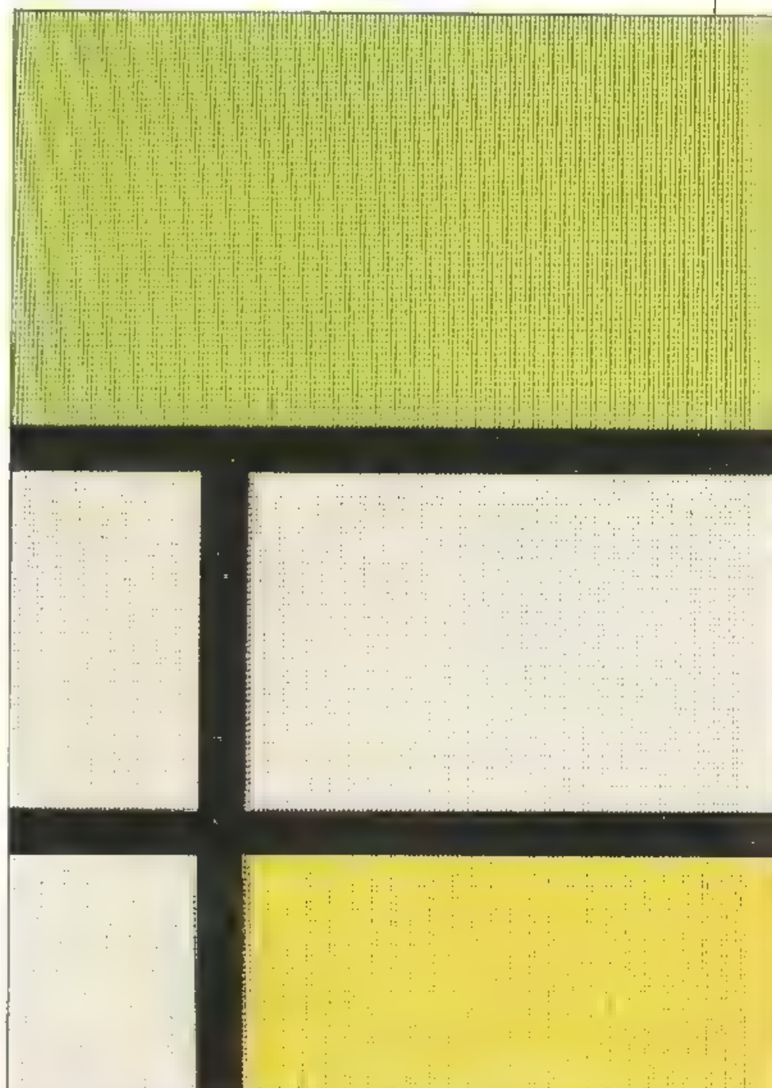


"MONDRIAN" PAINTING PROGRAM

00:01 second

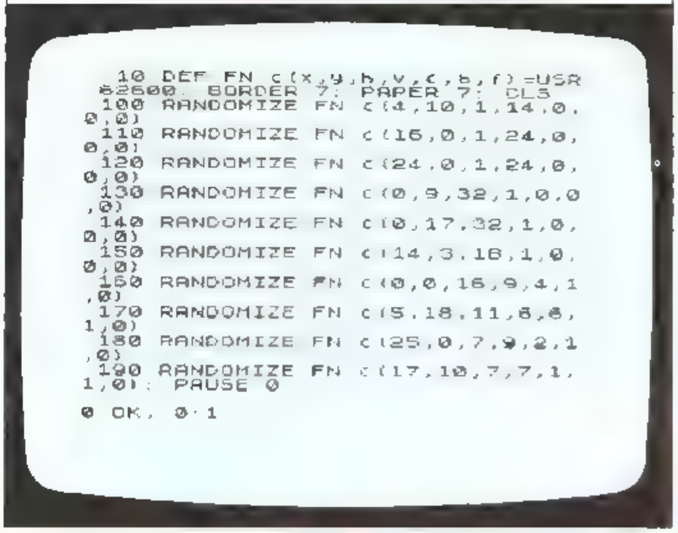
How the program works
The window paper routine draws black "lines" (single character-width boxes), and then fills areas of the screen with colour.

Line 10 defines the window paper routine.
Lines 100-150 draw the black "lines".
Lines 160-190 draw the coloured areas.
Line 190 also stops the program continuing until a key is pressed, so that the bottom two lines of the display are not lost.

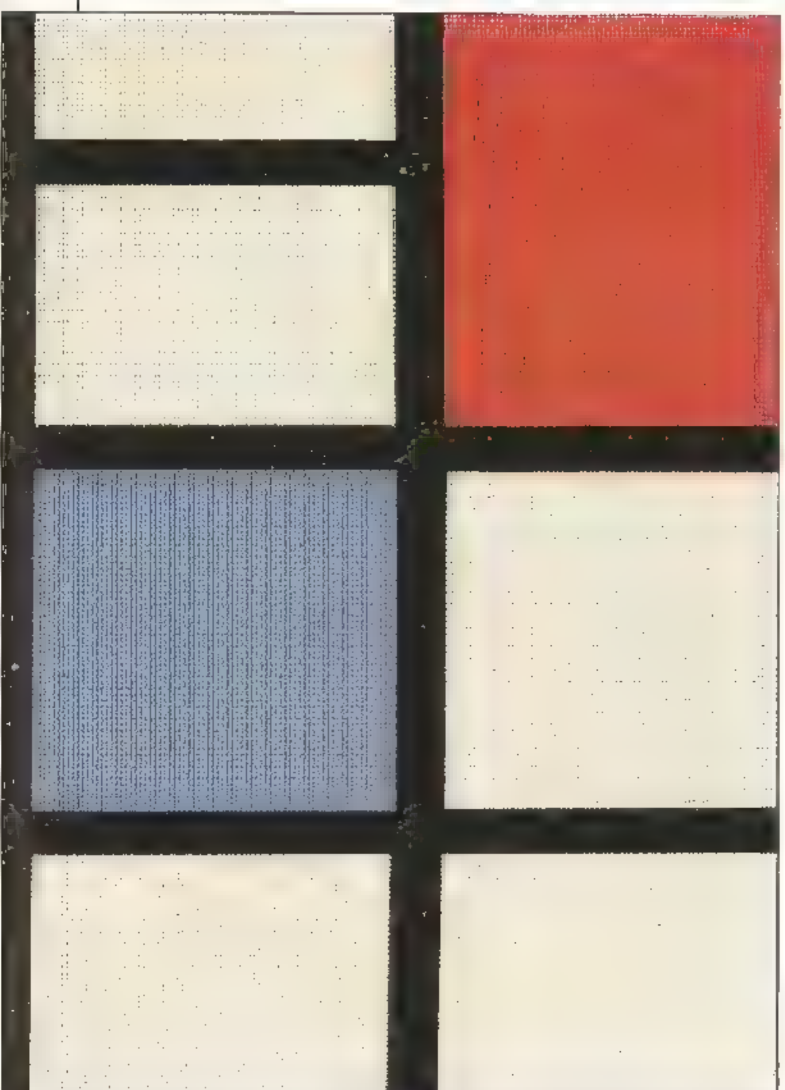


The "Mondrian" painting program demonstrates how by using only a single routine, you can produce quite an effective display.

"MONDRIAN" PAINTING PROGRAM



```
10 DEF FN c(x,y,h,v,c,b,f)=USR
62600: BORDER 7: PAPER 7: CLS
100 RANDOMIZE FN c(4,10,1,14,0,
0,0)
110 RANDOMIZE FN c(16,0,1,24,0,
0,0)
120 RANDOMIZE FN c(24,0,1,24,0,
0,0)
130 RANDOMIZE FN c(0,9,32,1,0,0
,0)
140 RANDOMIZE FN c(0,17,32,1,0,
0,0)
150 RANDOMIZE FN c(14,3,18,1,0,
0,0)
160 RANDOMIZE FN c(0,0,16,9,4,1
,0)
170 RANDOMIZE FN c(5,18,11,6,6,
1,0)
180 RANDOMIZE FN c(25,0,7,9,2,1
,0)
190 RANDOMIZE FN c(17,10,7,7,1,
1,0): PAUSE 0

0 OK, 0:1
```



FNc

WINDOW PAPER ROUTINE

Start address 62600 **Length** 150 bytes
What it does Changes the PAPER colour of a specified rectangular area of the screen.

Using the routine The routine works in the same way as the window ink routine, except that here the PAPER attributes are changed within the area specified. As before, it could be dangerous to go beyond the limits set for the parameters, so the sum of x and h should always be less than 32, and y and v together should be less than 24. This is because h and v are relative, not absolute, parameters, which means they are added to x and y respectively to produce the values actually plotted. Thus, if x is 15 and h is 20, then the right-hand edge of the paper area is column 35, which is off the screen.
As before, the routine operates over the whole Spectrum 32x24 character screen, not just over the normal 32x22 graphics area.

ROUTINE PARAMETERS	
DEF FNc(x,y,h,v,c,b,f)	
x,y	specify top left-handcorner of box area(x<32, y<24)
h,v	specify bottom right-hand corner of area (x+h<32, y+v<24)
c	specifies paper colour (0<=c<=7)
b	specifies bright (1=bright, 0=off)
f	specifies flash (1=flash, 0=off)

ROUTINE LISTING

```
7100 LET b=62600: LET l=145: LET
 z=0: RESTORE 7110
7101 FOR i=0 TO l-1: READ a
7102 POKE (b+i),a: LET z=z+a
7103 NEXT i
7104 LET z=INT (((z/l)-INT (z/l)
)*l)
7105 READ a: IF a<>z THEN PRINT
"??": STOP

7110 DATA 42,11,92,1,4
7111 DATA 0,9,86,1,8
7112 DATA 0,9,94,237,83
7113 DATA 22,245,9,86,9
7114 DATA 94,237,83,20,245
7115 DATA 9,126,230,7,203
7116 DATA 39,203,39,203,39
7117 DATA 50,19,245,9,126
7118 DATA 230,1,40,8,58
7119 DATA 19,245,246,64,50

7120 DATA 19,245,9,126,230
7121 DATA 1,40,8,58,19
7122 DATA 245,246,128,50,19
7123 DATA 245,237,91,22,245
7124 DATA 58,20,245,254,0
7125 DATA 200,58,21,245,254
7126 DATA 0,200,237,83,22
7127 DATA 245,123,230,24,203
7128 DATA 63,203,63,203,63
7129 DATA 246,88,103,123,230

7130 DATA 7,183,31,31,31
7131 DATA 31,130,111,58,20
7132 DATA 245,71,197,229,58
7133 DATA 21,245,71,126,230
7134 DATA 7,79,58,19,245
7135 DATA 177,119,35,16,244
7136 DATA 225,1,32,0,9
7137 DATA 193,16,230,201,0
7138 DATA 2,5,0,0,0
7139 DATA 19,0,0,0,0
```

ENLARGED TEXT

Doubling the size of Spectrum characters is quite straightforward in principle. Spectrum characters are drawn on a grid eight pixels by eight; they can be enlarged onto a 16x16 grid by the routine looking at each pixel of the 8x8 grid in turn. If a pixel is filled, then two pixels across and two pixels down are filled on the 16x16 grid. The diagram below gives an example of a character and its enlarged version.

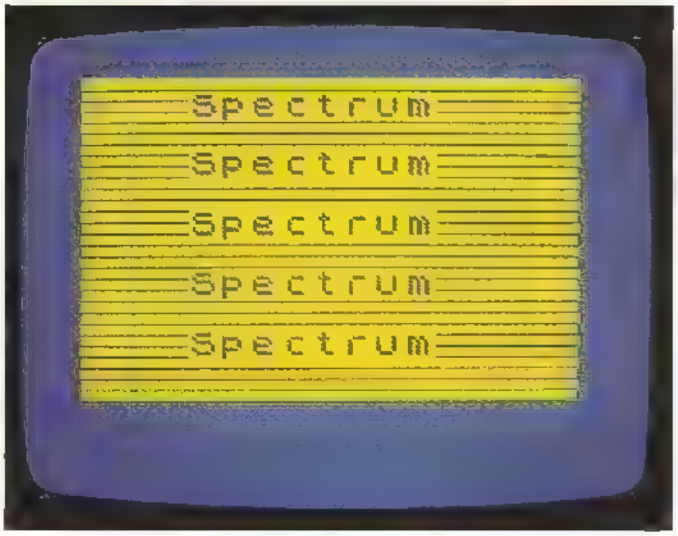
HOW A CHARACTER IS ENLARGED

Both routines use this method to enlarge a string of characters (text or graphics) and then print them on the screen at twice their normal size. The horizontal text routine, FNd, prints enlarged characters across the screen; the vertical text routine, FNe, prints the enlarged characters downwards.

The two demonstration programs below show how the routines are used. Both programs begin by defining the word "Spectrum" as the string (n$) to be enlarged by the routine, and both then POKE these characters into memory using a subroutine beginning at line 500. The string must always end with 13, the code for RETURN, to signal to the computer there are no more characters to be enlarged. The horizontal text program prints the string against a background of horizontal lines; the second program displays the vertical string six times, each time with a different coloured background, using the window paper routine.

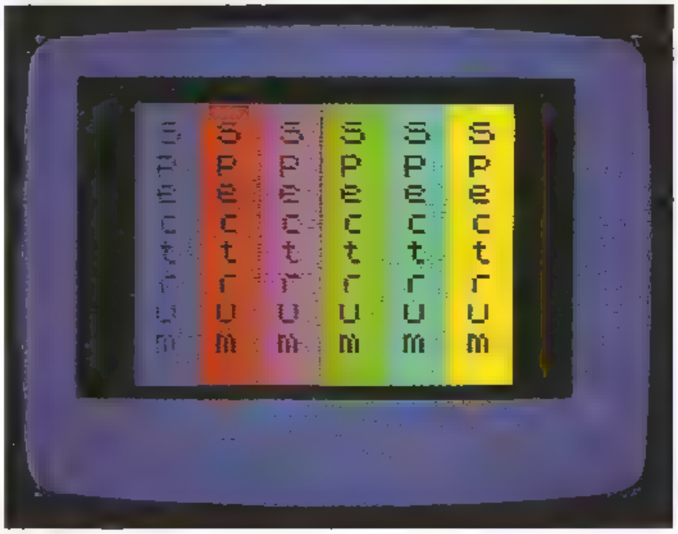
HORIZONTAL TEXT PROGRAM

```
  10 DEF FN d(x,y)=USR 62200
 100 BORDER 1: PAPER 6: INK 1: CLS
 110 FOR i=0 TO 13
 120 DRAW 255,0: DRAW 0,6
 130 DRAW -255,0: DRAW 0,6
 140 NEXT i
 150 DRAW 255,0
 160 LET n$="Spectrum"
 170 GO SUB 500
 180 FOR m=1 TO 22 STEP 4
 190 RANDOMIZE FN d(7,m)
 200 NEXT m
 499 PAUSE 0
 500 LET l=LEN (n$)
 505 LET k=62499
 510 FOR i=1 TO l
 520 LET n=CODE n$(i)
 530 POKE k+i,n
 540 NEXT i
 550 POKE k+i,13
 560 RETURN

0 OK, 0:1
```



VERTICAL TEXT PROGRAM

```
  10 DEF FN c(x,y,h,v,c,b,f)=USR 62600
  20 DEF FN e(x,y)=USR 61900
 100 BORDER 1: PAPER 0: INK [illegible]: CLS
 110 LET n$="Spectrum"
 120 GO SUB 500
 130 LET cl=1
 140 FOR x=5 TO 25 STEP 4
 150 RANDOMIZE FN c(x-1,2,4,19,cl,0,0)
 160 RANDOMIZE FN e(x,3)
 170 LET cl=cl+1
 180 NEXT x
 499 STOP
 500 LET l=LEN (n$): LET k=62499
 510 FOR i=1 TO l
 520 LET n=CODE n$(i)
 530 POKE k+i,n
 540 NEXT i
 550 POKE k+i,13
 560 RETURN

0 OK, 0:1
```



FNd

ENLARGED HORIZONTAL TEXT ROUTINE

Start address 62200 **Length** 220 bytes
What it does Displays a double-sized version of specified characters horizontally on the screen.

Using the routine Before using this routine, you must first use some BASIC lines to store in memory the text (n$) which you want to display. Lines 500-560 of the programs on the facing page provide an example.

The text is stored as a string in 100 bytes of memory from address 62500 to 62600. The routine continues printing characters from this location onwards until it reaches a RETURN message.

Note that with double-sized characters you are now restricted to 16 characters across the screen; longer strings are continued on the line below. To print a space in the text string, use the graphics blank character (above the 8 key) rather than the space key.

ROUTINE PARAMETERS

DEF FNd(x,y)	
x,y	specify position on screen from which text is to be printed (x<32, y<24)

ROUTINE LISTING

```
7150 LET b=62200: LET l=215: LET
 z=0: RESTORE 7160
7151 FOR i=0 TO l-1: READ a
7152 POKE (b+i),a: LET z=z+a
7153 NEXT i
7154 LET z=INT (((z/l)-INT (z/l)
)*l)
7155 READ a: IF a<>z THEN PRINT
"??": STOP

7160 DATA 42,11,92,1,4
7161 DATA 0,9,86,1,8
7162 DATA 0,9,94,237,83
7163 DATA 240,243,62,99,71
7164 DATA 33,36,244,34,244
7165 DATA 243,197,237,91,240
7166 DATA 243,62,30,186,242
7167 DATA 37,243,22,0,28
7168 DATA 28,237,83,240,243
7169 DATA 62,20,187,250,111

7170 DATA 243,42,244,243,126
7171 DATA 35,34,244,243,254
7172 DATA 31,250,111,243,254
7173 DATA 144,242,111,243,214
7174 DATA 32,1,8,0,42
7175 DATA 54,92,36,9,61
7176 DATA 32,252,34,242,243
7177 DATA 123,230,24,246,64
7178 DATA 103,123,230,7,183
7179 DATA 31,31,31,31,130

7180 DATA 111,34,238,243,205
7181 DATA 113,243,58,241,243
7182 DATA 60,60,50,241,243
7183 DATA 193,16,164,201,193
7184 DATA 201,17,206,243,6
7185 DATA 32,62,0,18,19
7186 DATA 16,252,237,91,242
7187 DATA 243,33,206,243,6
7188 DATA 8,197,26,1,2
7189 DATA 4,197,23,245,203

7190 DATA 22,241,203,22,16
7191 DATA 247,35,193,13,32
7192 DATA 241,43,125,245,43
7193 DATA 126,35,35,119,35
7194 DATA 241,119,35,19,193
7195 DATA 16,220,42,238,243
7196 DATA 17,206,243,14,2
7197 DATA 229,6,8,26,119
7198 DATA 35,19,26,119,19
7199 DATA 43,36,16,245,225
7200 DATA 62,32,133,111,48
7201 DATA 4,62,8,132,103
7202 DATA 13,32,228,201,0
7203 DATA 0,0,0,0,0
```

FNe

ENLARGED VERTICAL TEXT ROUTINE

Start address 61900 **Length** 215 bytes
What it does Displays a double-sized version of specified characters vertically on the screen.

Using the routine This routine works in the same way as the horizontal text routine, but prints text down the screen instead of across it. The same BASIC subroutine is needed to store the text string (lines 500-560 of the demonstration programs on the facing page). Remember to put the string into memory before calling the routine.

Since each character is twice its normal size, only 12 characters down are shown in a column. The routine displays only one vertical line of text, and does not continue a message across to the next column. To display a message longer than 12 characters, call the routine again for each new column of text. To obtain a space in the text use the graphics blank character (above key 8).

ROUTINE PARAMETERS

DEF FNe(x,y)	
x,y	specify position on screen from which text is to be printed (x<32, y<24)

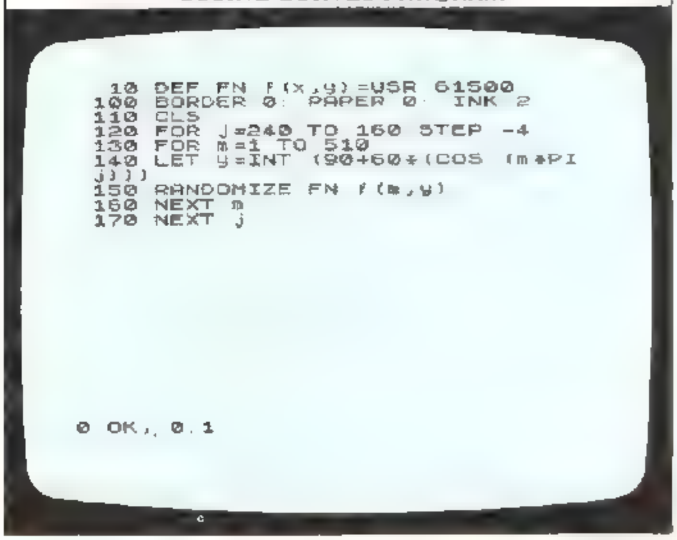
ROUTINE LISTING

```
7250 LET b=61900: LET l=210: LET
 z=0: RESTORE 7260
7251 FOR i=0 TO l-1: READ a
7252 POKE (b+i),a: LET z=z+a
7253 NEXT i
7254 LET z=INT (((z/l)-INT (z/l)
)*l)
7255 READ a: IF a<>z THEN PRINT
"??": STOP

7260 DATA 42,11,92,1,4
7261 DATA 0,9,86,1,8
7262 DATA 0,9,94,237,83
7263 DATA 191,242,62,99,71
7264 DATA 33,36,244,34,195
7265 DATA 242,197,237,91,191
7266 DATA 242,62,30,186,242
7267 DATA 244,241,195,62,242
7268 DATA 62,20,187,250,62
7269 DATA 242,42,195,242,126

7270 DATA 35,34,195,242,254
7271 DATA 31,250,62,242,254
7272 DATA 144,242,62,242,214
7273 DATA 32,1,8,0,42
7274 DATA 54,92,36,9,61
7275 DATA 32,252,34,193,242
7276 DATA 123,230,24,246,64
7277 DATA 103,123,230,7,183
7278 DATA 31,31,31,31,130
7279 DATA 111,34,189,242,205

7280 DATA 64,242,58,191,242
7281 DATA 60,60,50,191,242
7282 DATA 193,16,169,201,193
7283 DATA 201,17,157,242,6
7284 DATA 32,62,0,18,19
7285 DATA 16,252,237,91,193
7286 DATA 242,33,157,242,6
7287 DATA 8,197,26,1,2
7288 DATA 4,197,23,245,203
7289 DATA 22,241,203,22,16

7290 DATA 247,35,193,13,32
7291 DATA 241,43,126,245,43
7292 DATA 126,35,35,119,35
7293 DATA 241,119,35,19,193
7294 DATA 16,220,42,189,242
7295 DATA 17,157,242,14,2
7296 DATA 229,6,8,26,119
7297 DATA 35,19,26,119,19
7298 DATA 43,36,16,245,225
7299 DATA 62,32,133,111,48
7300 DATA 4,62,8,132,103
7301 DATA 13,32,228,201,0
7302 DATA 125,0,0,0,0
```

PICTURES WITH POINTS 1

The Spectrum ROM routine which is called by the BASIC command PLOT to draw single points is also used by the BASIC DRAW and CIRCLE commands. The point plot routine given here, FNg, is used in the same way, both to plot points on the screen, and to provide the basis for the other drawing routines in this book, including routines for lines, boxes and circles.

COSINE CURVES PROGRAM



The demonstration programs on this page may seem slower than you would expect. This is not due to the speed of the routine, but because the BASIC program is switching to machine code for each single point and then returning to BASIC. Later drawing routines, which call the point-plot routine from machine code, give a better indication of the routine's true speed. The programs here show only the difference in speed between the BASIC commands RANDOMIZE and PLOT.

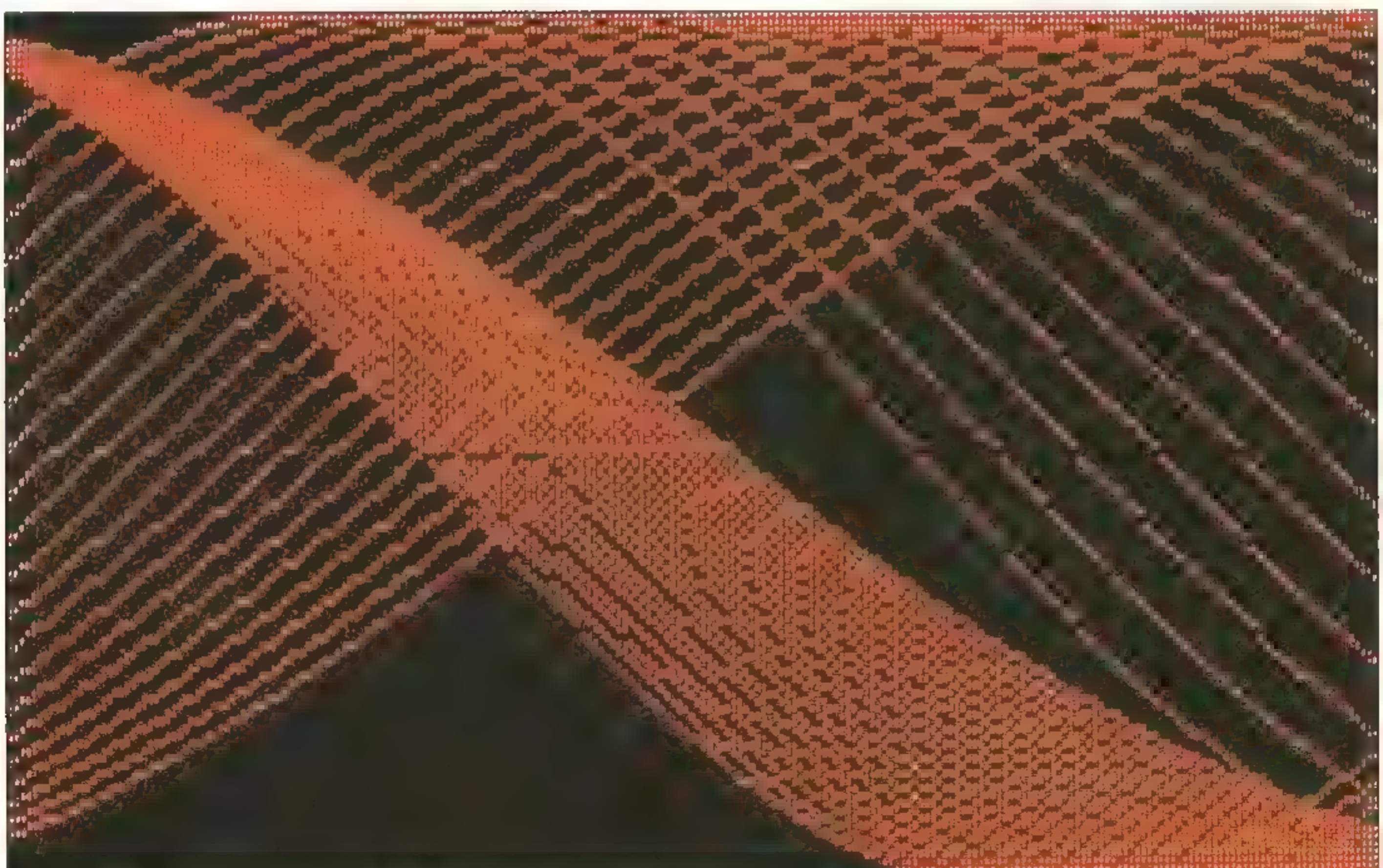
The planet program plots random points on horizontal lines which begin and end on the circumference of a circle. There is an increasing probability of a point being plotted towards the right of each line (line 540). The series of exponent curves are produced by varying the horizontal co-ordinate, x.

COSINE CURVES PROGRAM

12:30 **minutes**

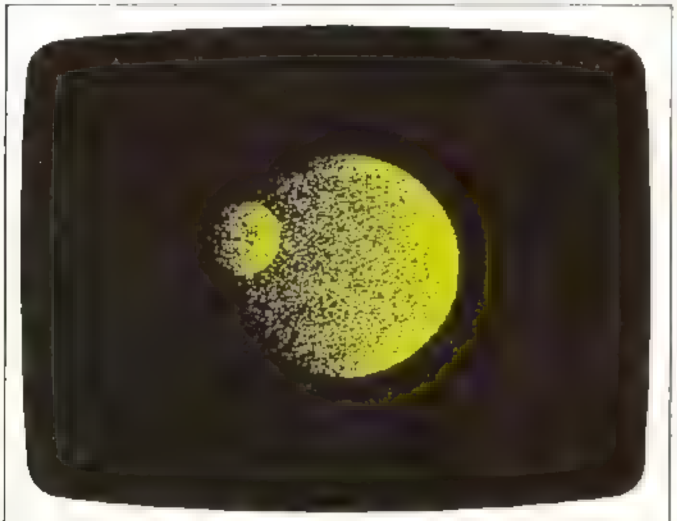
How the program works
Over 10,000 points are plotted in a series of cosine waves.

Line 10 defines the function.
Line 130 sets the horizontal start co-ordinate of each curve.
Line 140 calculates the y co-ordinate (each curve is a slightly different function, since j varies for each curve).
Line 150 plots a point at m,y.



PLANET PROGRAM

```
  10 DEF FN f(x,y)=USR 61500
 100 BORDER 0: PAPER 0: INK 4: C
LS
 110 LET r=60: LET xc=127: LET y
c=88
 120 GO SUB 500
 130 LET r=20: LET xc=75: LET yc
=100
 140 GO SUB 500
 499 STOP
 500 FOR y=-r TO r
 510 LET x1=INT (SQR (r*r-y*y))
 520 FOR x=-x1 TO x1
 530 LET n=INT (RND*(1)*x1+2)+1
 540 IF n<x1+x THEN RESTORE FN f
(x+xc,y+yc)
 550 NEXT x
 560 NEXT y
 570 RETURN


0 OK, 0:1
```



Line 130 raises x to the power n to determine the point for plotting, z. Line 160 plots the curve again, subtracting co-ordinates from an initial value.

EXPONENT CURVES PROGRAM

```
  10 DEF FN f(x,y)=USR 61500
 100 BORDER 6: PAPER 6: INK 0: C
LS
 110 FOR n=1.19 TO 1.80 STEP 0.0
1
 120 FOR x=0 TO 22 STEP 0.5
 130 LET z=INT (x↑n)
 140 LET y=INT (x*8)
 150 RANDOMIZE FN f(z,y)
 160 RANDOMIZE FN f(255-z,168-y)
 170 NEXT x
 180 NEXT n


0 OK, 0:1
```

FNf

POINT-PLOT ROUTINE

Start address 61500 **Length** 65 bytes
What it does Plots a single point on the screen.

Using the routine The point-plot routine uses pixel rather than character co-ordinates. Pixel co-ordinates are calculated from the bottom left-hand corner of the screen, unlike character co-ordinates which start on the Spectrum from the top left-hand corner. Thus, points are calculated on the screen from 0 to 175 vertically upwards, and from 0 to 255 horizontally: point (255,175) is the top right-hand corner of the screen, for example. Note that routines in this book which use pixel points will not go over the text area of the screen (the bottom two lines of the screen) since the point (0,0) is actually above these two lines.

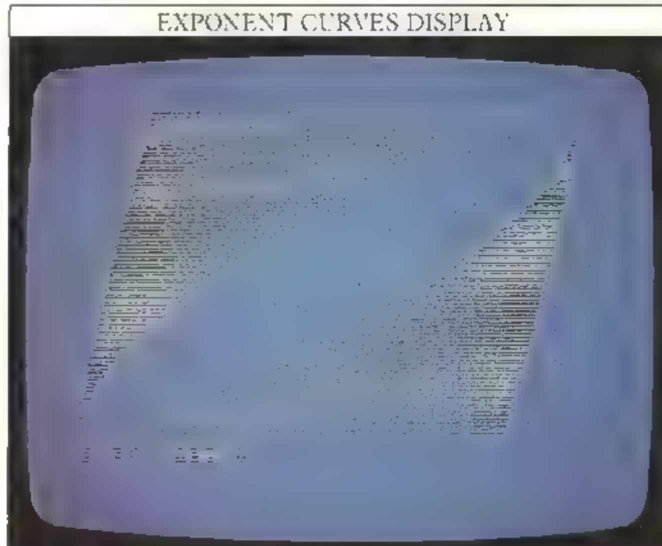
ROUTINE PARAMETERS

DEF FNf(x,y)	
x,y	specify pixel position at which point is to be plotted ($x<256, y<176$)

ROUTINE LISTING

```
7350 LET b=61500: LET l=60: LET
z=0: RESTORE 7360
7351 FOR i=0 TO l-1: READ a
7352 POKE (b+i),a: LET z=z+a
7353 NEXT i
7354 LET z=INT (((z/l)-INT (z/l)
)*l)
7355 READ a: IF a<>z THEN PRINT
"??": STOP

7360 DATA 42,11,92,1,4
7361 DATA 0,9,86,14,8
7362 DATA 9,94,62,175,147
7363 DATA 216,95,167,31,55
7364 DATA 31,167,31,171,230
7365 DATA 248,171,103,122,7
7366 DATA 7,7,171,230,199
7367 DATA 171,7,7,111,122
7368 DATA 230,7,71,4,62
7369 DATA 254,15,16,253,6
7370 DATA 255,168,71,126,176
7371 DATA 119,201,0,0,0
7372 DATA 24,0,0,0,0
```

EXPONENT CURVES DISPLAY



PICTURES WITH POINTS 2

The displays on the previous page used only a simple BASIC listing and a single routine, the point-plot routine. There is no reason why you should not combine routines together to produce far more complex displays, as demonstrated here.

The cityscape program

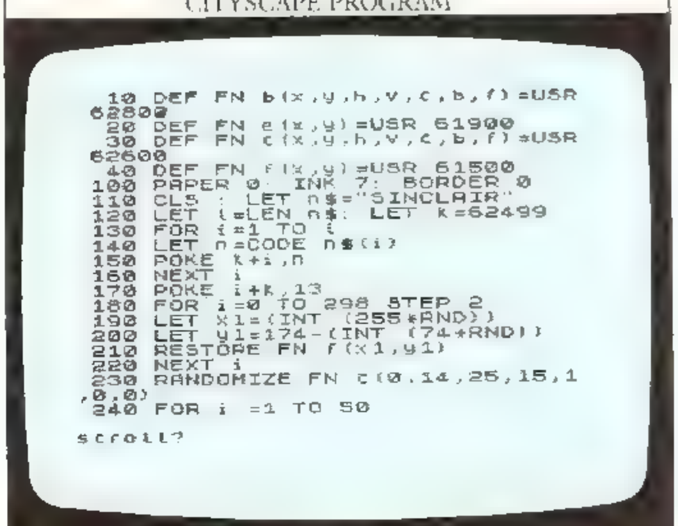
The large display on this page is produced by a single program, the cityscape program. The program combines plotted points with three other routines to produce the display.

A total of four routines is used in this program. The skyscrapers are drawn by the window paper routine, FNb; the vertical text routine (FNe) is used to draw the word SINCLAIR, printed in blue by the window ink routine.

The effect of a crowded group of skyscrapers is achieved by drawing coloured windows at random. The effect of random heights but a constant base line is achieved by making all the windows end on the bottom line of the screen. This is done by subtracting the start y co-ordinate (y1) from 25, the total number of vertical text characters on the screen plus one.

The vertical text routine, which is used to print the word "SINCLAIR", must have the letters which are to be displayed placed in memory before the routine is called. Lines 110-170 take the characters from the word one at a time and POKE them into memory ready to be used by the routine. As before, the final character entered is 13, the ASCII code for carriage return. You will remember that the routine requires the co-ordinates of the top left-hand character (x,y) as well as the stored text string in order to print the text. The window ink routine is used in line 360 to give a blue colour to the area over which the text is to be printed.

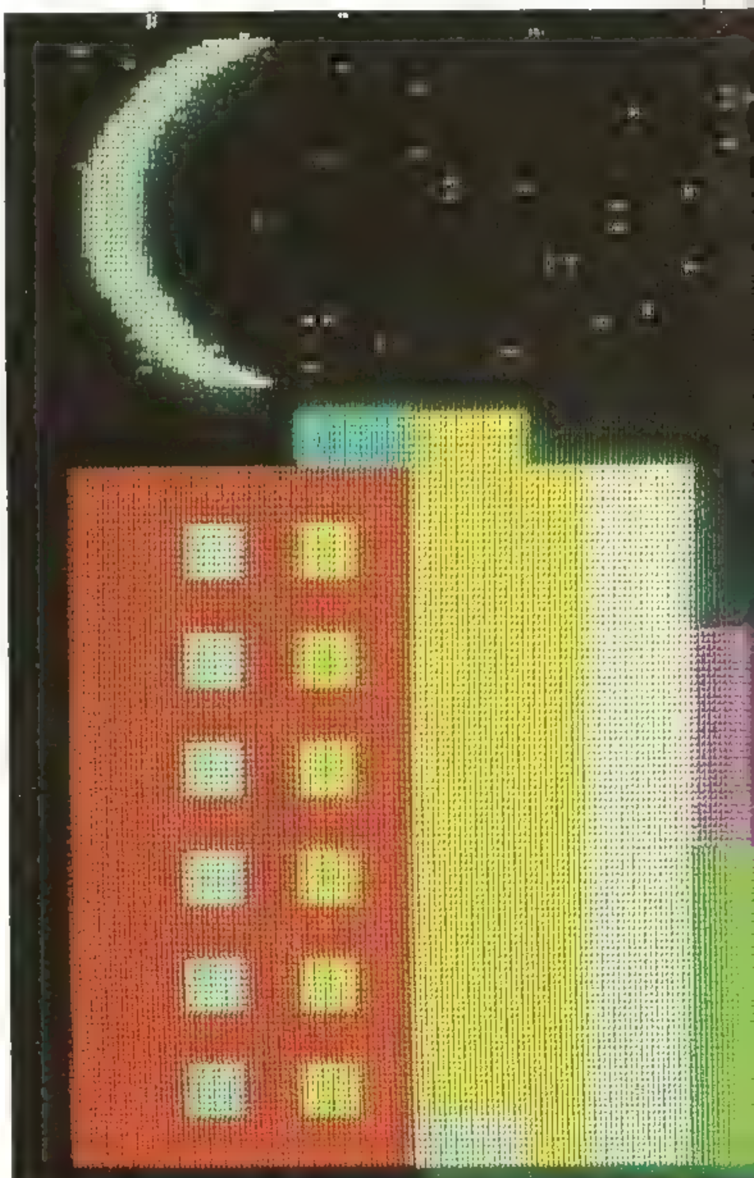
CITYSCAPE PROGRAM



```
  10 DEF FN b(x,y,h,v,c,b,f)=USR
 62800
  20 DEF FN e(x,y)=USR 61900
  30 DEF FN c(x,y,h,v,c,b,f)=USR
 62600
  40 DEF FN f(x,y)=USR 61500
 100 PAPER 0  INK 7: BORDER 0
 110 CLS : LET n$="SINCLAIR"
 120 LET l=LEN n$: LET k=62499
 130 FOR i=1 TO l
 140 LET n=CODE n$(i)
 150 POKE k+i,n
 160 NEXT i
 170 POKE i+k,13
 180 FOR i=0 TO 298 STEP 2
 190 LET x1=(INT (255*RND))
 200 LET y1=174-(INT (74*RND))
 210 RESTORE FN f(x1,y1)
 220 NEXT i
 230 RANDOMIZE FN c(0.14,25,15,1
,0.0)
 240 FOR i =1 TO 50

scroll?
```

Line 210 calls the point-plot routine to plot the stars. Random co-ordinates are chosen in lines 190 and 200 for each star. The moon is drawn in lines 530-560 by using semicircles nested inside each other, using the Spectrum BASIC DRAW command. A later routine in this book will enable you to produce circles using machine code. Finally, the meteor is drawn as a series of straight lines (lines 490-510), again using BASIC.

You will notice from the listing for this program that a convention has been used for all the listings in this book. Lines numbered from 10 to 90 are used for the function definitions, while lines 100 onwards are used for the main listing. You can thus see clearly which machine-code routines have been used for each program, as they are placed at the beginning of the listing.



CITYSCAPE PROGRAM CONTD.

```
 250 RANDOMIZE
 260 LET h1=2+INT (RND*4)
 270 LET y1=10+INT (RND*15)
 280 LET v1=25-y1
 290 LET x1=INT (RND*27)
 300 LET c1=2+INT (RND*6)
 310 RANDOMIZE FN c(x1;y1,h1,v1,
c1,0,0)
 320 NEXT i
 330 RANDOMIZE FN c(16,9,1,15,4,
1,0)
 340 RANDOMIZE FN c(17,6,2,18,4,
1,1)
 350 RANDOMIZE FN c(19,10,1,14,4
,1,0)
 360 RANDOMIZE FN b(17,6,2,18,1,
0,1)
 370 RANDOMIZE FN e(17,6)
 380 RANDOMIZE FN c(1,11,5,14,2,
0,0)
 390 FOR i=12 TO 22 STEP 2
 400 RANDOMIZE FN c(2,i,1,1,7,1,

scroll?
```

CITYSCAPE PROGRAM CONTD.

```
0)
 410 RANDOMIZE FN c(4,i,1,1,6,1,
0)
 420 NEXT i
 430 RANDOMIZE FN c(23,18,8,7,3,
0,0)
 440 FOR i=24 TO 30 STEP 2
 450 RANDOMIZE FN c(i,19,1,1,4,1
,0)
 460 RANDOMIZE FN c(i,21,1,1,4,1
,0)
 470 NEXT i
 480 INK 6: BRIGHT 1
 490 FOR i=-6 TO 6 STEP 2
 500 PLOT 250,165-i
 510 DRAW -70,i-40: NEXT i
 520 INK 7
 530 FOR i=0 TO 7
 540 PLOT 20+i,150
 550 DRAW 0,-50,0.6*PI
 560 NEXT i
 570 PAUSE 0

0 OK, 0:1
```

CITYSCAPE PROGRAM

00:15 **seconds**

This program calls four routines, all of which must be present in memory before RUNning the program:

- window ink routine (FNb) page 11
- window paper routine (FNc) page 13
- enlarged vertical text routine (FNe) page 15
- point plot routine (FNf) page 17

How the program works

Lines 10-40 define the routines.

Line 110 defines the text string.

Line 120 sets up a loop to POKE characters into memory.

Line 130 POKEs a single character into memory.

Line 170 POKEs ASCII code 13 into memory.

Lines 180-220 print stars at random points in the top 74 pixel rows of the screen.

Lines 240-320 set up values for the window paper boxes and draw them — a total of 50 boxes.

Lines 330-380 draw the "buildings" with windows which appear in front of the paper boxes (lines 340 and 360 print flashing boxes).

Lines 480-510 draw the comet.

Lines 520-570 draw the moon (a series of semicircles).

LINE GRAPHICS 1

Lines are drawn in BASIC on the Spectrum using the command DRAW. This command uses relative co-ordinates, that is, the command is followed by co-ordinates which specify the distance from the current plot position. A line is then drawn from this point to the specified point. It isn't always as simple as it may seem to calculate this horizontal and vertical increment from the current plot position.

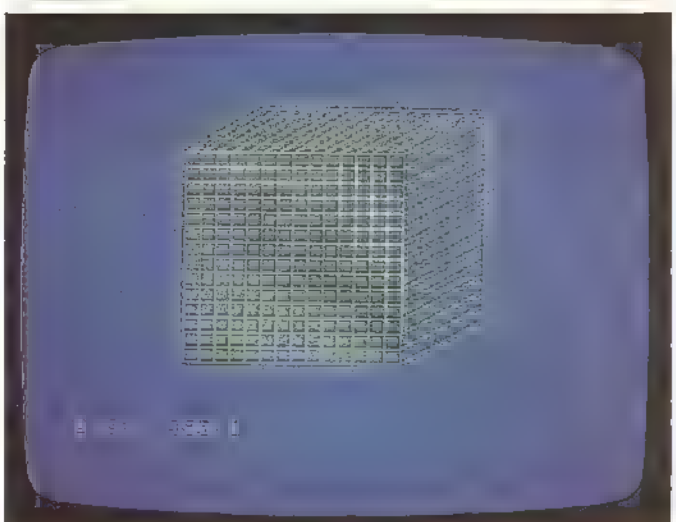
The line-draw routine

The routine on this page, FNg, offers an alternative to Spectrum DRAW for drawing straight lines. This routine is faster than the DRAW command, and uses absolute, rather than relative co-ordinates. Thus, you no longer have to worry about calculating distances from the current plot position.

The line-draw routine contains some error-trapping to prevent the Spectrum crashing if the routine is called

CUBE PROGRAM

```
  10 DEF FN g(x,y,p,q)=USR 60700
 100 BORDER 1: PAPER 1: INK 6
 110 CLS
 120 FOR j=0 TO 116 STEP 8
 130 RANDOMIZE FN g(56,24+j,168,
24+j)
 140 RANDOMIZE FN g(56+j,24,56+j
,136)
 150 RANDOMIZE FN g(56+j,136,96+
j,160)
 160 RANDOMIZE FN g(168,24+j,208
,48+j)
 170 NEXT j
 180 RANDOMIZE FN g(96,160,208,1
60)
 190 RANDOMIZE FN g(208,160,208,
48)


0 OK, 0:1
```



SUNSET PROGRAM

```
  10 DEF FN t(x,y,h,v,c,b,f)=USR
 62600
  20 DEF FN g(x,y,p,q)=USR 60700
 100 BORDER 0: PAPER 1: INK 6: C
LS
 110 RANDOMIZE FN t(0,0,32,17,2,
0,0)
 120 FOR j=40 TO 174 STEP 12
 130 RANDOMIZE FN g(0,j,128,40)
 140 RANDOMIZE FN g(255,j,128,40
)
 150 NEXT j
 160 FOR j=6 TO 256 STEP 12
 170 RANDOMIZE FN g(j,175,128,40
)
 180 NEXT j
 190 FOR j=36 TO 0 STEP -3
 200 RANDOMIZE FN g((10+j*3),j,(
246-j*3),j)
 210 NEXT j


0 OK, 0:1
```

with off-screen co-ordinates. This makes it easier to devise complex graphics displays using a trial and error method, since there is less danger of losing both program and routine if off-screen co-ordinates are entered.

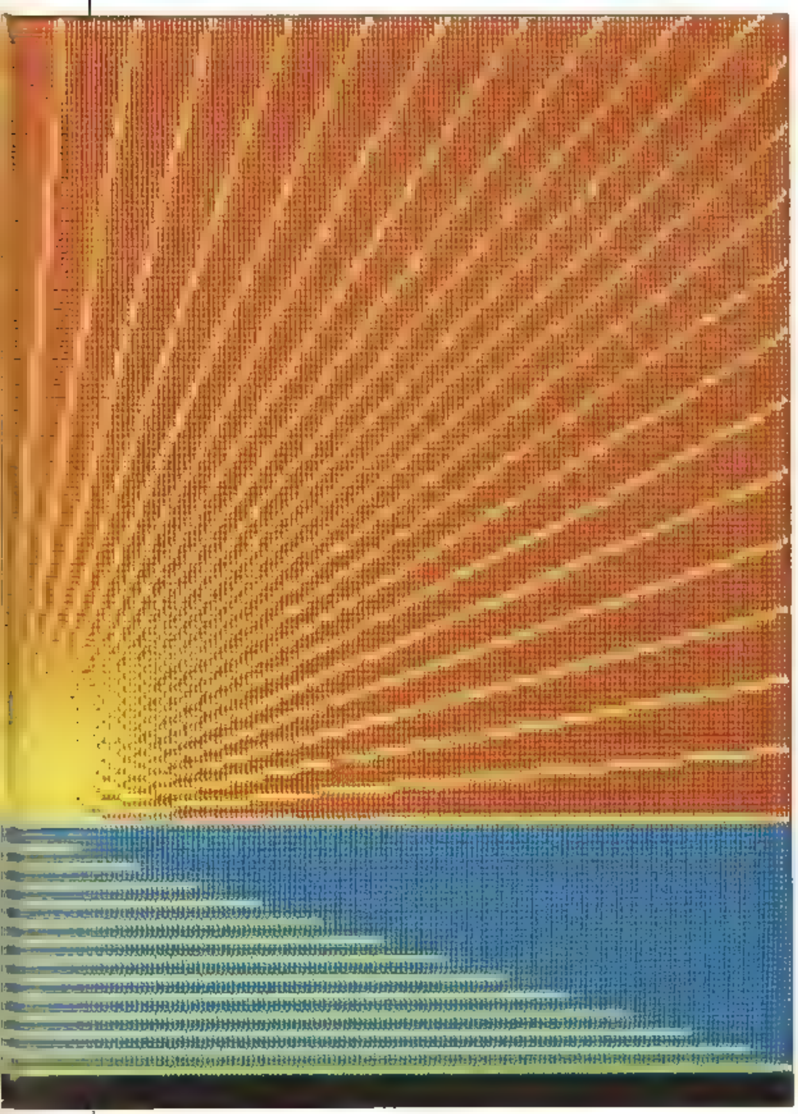
The cube display on page 20 is formed from a series of lines. The program is very simple. The line draw routine draws four lines repeatedly in the loop from lines 120 to 170: two to draw the grid pattern, and two for the perspective effect. Lines 180 and 190 specify the two lines which complete the cube shape.

SUNSET PROGRAM

00:03 **seconds**

How the program works
Yellow lines are drawn radiating from a point at the centre of the screen to points on the edge. Horizontal lines are then added to create a reflection effect.

Line 110 sets the blue colour in the bottom part of the screen using the window paper routine.
Lines 130 and 140 draw the horizon.
Lines 160-180 draw the yellow lines in the sky.
Lines 190-210 draw the horizontal lines on the lower half of the screen.



FNg

LINE-DRAW ROUTINE

Start address 60700 **Length** 205 bytes
What it does Draws a line between two specified points.

Using the routine This routine draws a line joining any two pixel points on the screen. Although the Spectrum already has a line draw routine available in BASIC, the version given here is much faster, and uses absolute rather than relative co-ordinates. Ths means that co-ordinates p,q represent the position of the end point of the line, not the horizontal and vertical increment from x,y.
The routine will usually work if off-screen points are specified, but for safety some error-trapping has been incorporated. If you attempt to plot lines off the screen, you will see an "Integer out of range" message, unless the value you have entered is more than 255 pixels off the screen; in this case the routine will probably crash.

ROUTINE PARAMETERS	
DEF FNg(x,y,p,q)	
x,y	specify start position of line (x<256,y<176)
p,q	specify end position of line (p<256,q<176)

ROUTINE LISTING

```
7400 LET b=60700: LET l=210: LET
 z=0: RESTORE 7410
7401 FOR i=0 TO l-1: READ a
7402 POKE (b+i),a: LET z=z+a
7403 NEXT i
7404 LET z=INT (((z/l)-INT (z/l)
)*l)
7405 READ a: IF a<>z THEN PRINT
"??": STOP

7410 DATA 42,11,92,1,4
7411 DATA 0,9,86,14,8
7412 DATA 9,94,237,83,26
7413 DATA 237,205,226,237,94
7414 DATA 42,26,237,217,229
7415 DATA 217,237,115,176,237
7416 DATA 1,1,1,122,148
7417 DATA 210,70,237,6,255
7418 DATA 237,68,87,123,149
7419 DATA 210,80,237,14,255

7420 DATA 237,68,95,122,187
7421 DATA 48,10,106,237,67
7422 DATA 177,237,175,71,195
7423 DATA 107,237,178,202,167
7424 DATA 237,107,90,237,67
7425 DATA 177,237,14,0,99
7426 DATA 123,31,133,218,118
7427 DATA 237,188,218,120,237
7428 DATA 148,87,217,237,91
7429 DATA 177,237,195,132,237

7430 DATA 87,197,217,209,42
7431 DATA 26,237,123,133,95
7432 DATA 122,60,132,218,164
7433 DATA 237,202,167,237,61
7434 DATA 87,237,83,26,237
7435 DATA 205,179,237,217,122
7436 DATA 29,32,205,195,167
7437 DATA 237,202,147,237,237
7438 DATA 123,175,237,217,225
7439 DATA 217,201,181,214,1

7440 DATA 1,62,175,147,218
7441 DATA 249,36,95,167,31
7442 DATA 55,31,167,31,171
7443 DATA 230,248,171,103,122
7444 DATA 7,7,7,171,230
7445 DATA 199,171,7,7,111
7446 DATA 122,230,7,71,4
7447 DATA 62,254,15,16,253
7448 DATA 6,255,168,71,126
7449 DATA 176,119,201,229,197
7450 DATA 205,179,237,193,225
7451 DATA 9,86,9,201,0
7452 DATA 192,0,0,0,0
```

LINE GRAPHICS 2

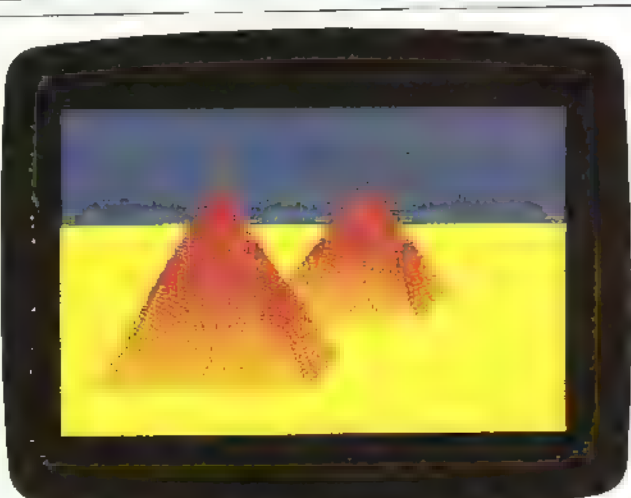
Line-drawing routines are ideal for producing interference patterns. These are produced when a series of lines or points are drawn so close together that what is produced is neither separate lines nor a complete solid, but a pattern.

The pyramid program below shows interference patterns at work. Each pyramid is drawn by a subroutine beginning at line 500, which draws lines from the top of the pyramid (the fixed point tx,ty) to points along a horizontal base line (by). Only the base x co-ordinate (x) is varied within the loop. Interference patterns are seen from near the top of the pyramid (where the lines no longer have the appearance of a solid figure) to a point towards the base of the pyramid (where lines are beginning to be seen distinctly).

The line pattern program demonstrates a related phenomenon. Here the line-draw routine (called in lines 140-170) has the paradoxical effect of producing

PYRAMID PROGRAM

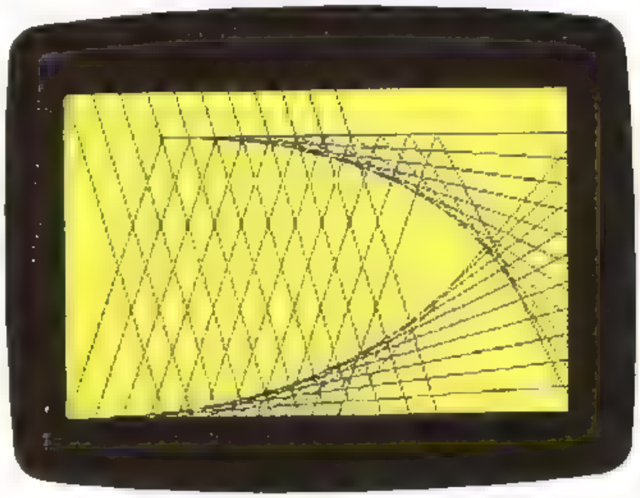
```
  10 DEF FN b(x,y,h,v,c,b,f)=USR
 62800
  20 DEF FN c(x,y,h,v,c,b,f)=USR
 62600
  30 DEF FN g(x,y,p,q)=USR 60700
 100 BORDER 0: PAPER 1: INK 2
 110 CLS
 120 RANDOMIZE FN c(0,8,31,14,6,
0,0)
 130 LET tx=60: LET ty=136: LET
by=24: LET a=16: LET b=128
 140 GO SUB 500
 150 LET tx=188: LET by=60: LET
a=102: LET b=182
 160 GO SUB 500: STOP
 500 FOR x=a TO b STEP 2
 510 RANDOMIZE FN g(tx,ty,x,by)
 520 NEXT x
 530 FOR x=b TO b+16
 540 RANDOMIZE FN g(tx,ty,x,by)
 550 LET by=by+1
 560 NEXT x: RETURN

0 OK, 0:1
```

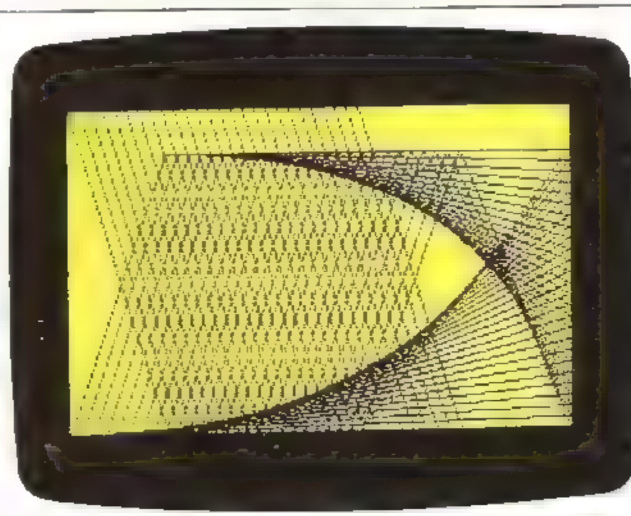


LINE PATTERN PROGRAM

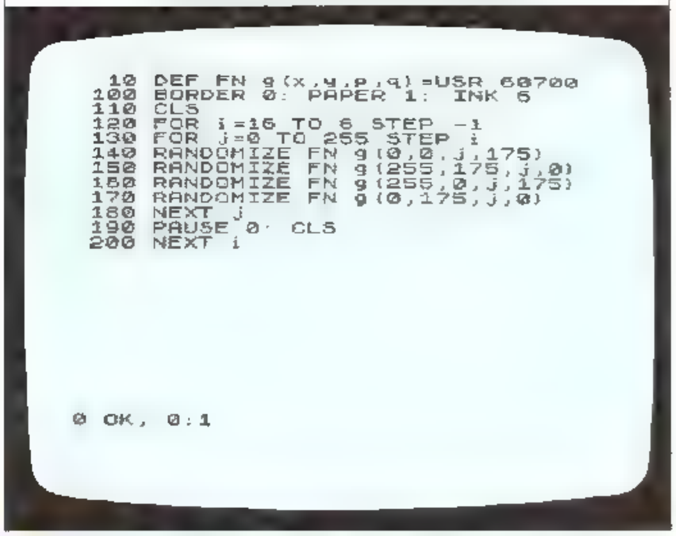
```
  10 DEF FN g(x,y,p,q)=USR 60700
 100 BORDER 0: PAPER 6: INK 0
 110 CLS
 120 FOR i=14 TO 2 STEP -1
 130 FOR j=0 TO 150 STEP i
 140 RANDOMIZE FN g(50+j,0,j,175
)
 150 RANDOMIZE FN g(255,j,j,0)
 160 RANDOMIZE FN g(50+j,150,255
,150-j)
 170 RANDOMIZE FN g(50+j,150,j,0
)
 180 NEXT j
 190 PAUSE 0: CLS
 200 NEXT i

0 OK, 0:1
```

LINE PATTERN DISPLAYS



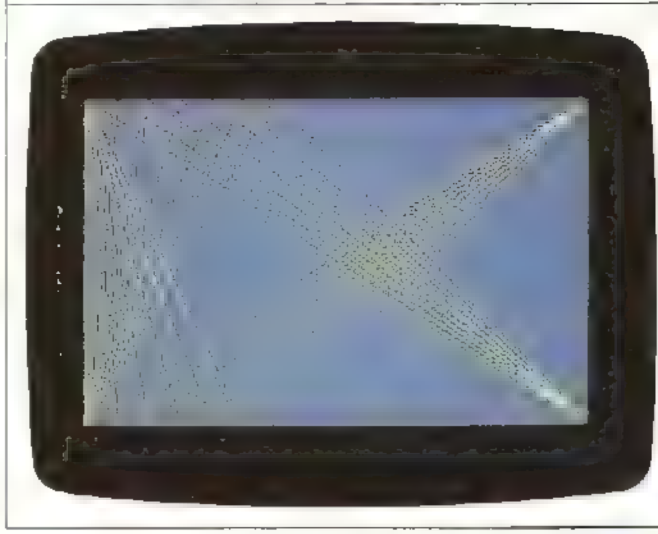


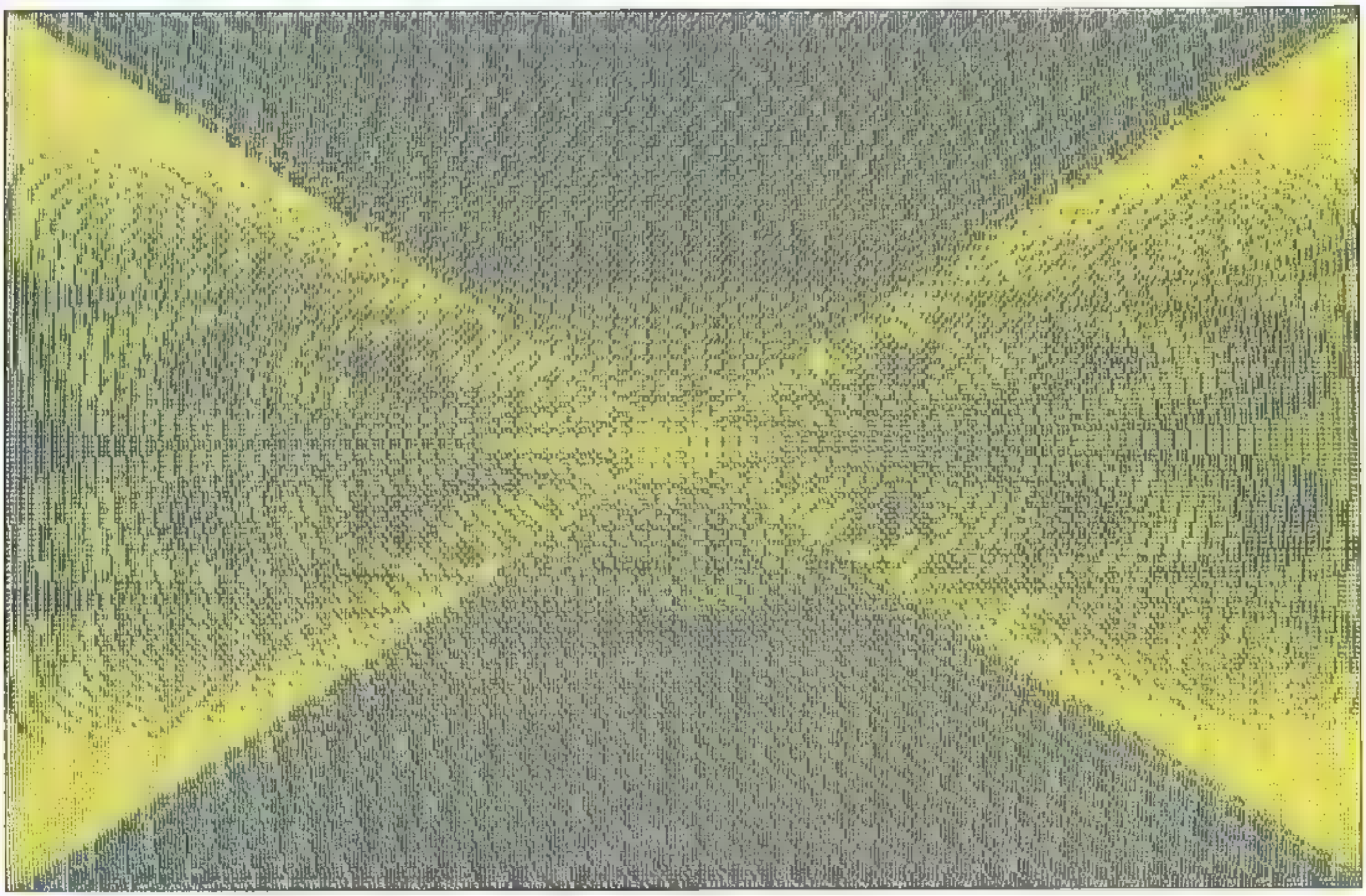
LINE INTERFERENCE PROGRAM



```
 10 DEF FN g(x,y,p,q)=USR 60700
100 BORDER 0: PAPER 1: INK 6
110 CLS
120 FOR i=16 TO 6 STEP -1
130 FOR j=0 TO 255 STEP i
140 RANDOMIZE FN g(0,0,j,175)
150 RANDOMIZE FN g(255,175,j,0)
160 RANDOMIZE FN g(255,0,j,175)
170 RANDOMIZE FN g(0,175,j,0)
180 NEXT j
190 PAUSE 0: CLS
200 NEXT i

0 OK, 0:1
```

LINE INTERFERENCE DISPLAY





curves. The series of horizontal and vertical lines in sequence produces the curve effect. As the lines become closer together, with each successive display, a better effect is obtained.

The line interference program shows how a program similar to the line pattern program can produce interference patterns simply by increasing the number of lines plotted on the screen, from 150 to 255 (the variable i in line 130).

LINE INTERFERENCE PROGRAM

00:06 seconds

How the program works
Patterns are produced by drawing lines from each corner of the screen to the opposite screen edge.
Line 120 sets up the first loop, to draw complete displays.
Line 130 sets up the second, inner loop, which calls the routines 255 times to draw a single display
Lines 140-170 call the line routine to draw four lines.
Line 190 waits for a key to be pressed before clearing the screen and beginning the next display.

TRIANGLES

A triangle shape is useful as the basis of all kinds of graphics displays. Pyramids, mountains, trees and bushes can all be formed from a triangular shape; even the spotlight display on this page is drawn with triangles. However, Spectrum BASIC does not have a single-statement triangle command.

The triangle routine, FNi, enables you to draw triangles quickly and painlessly. Like the line-draw routine (FNg), on which it is based, the triangle routine uses absolute rather than relative co-ordinates; this makes complex graphics displays easier to program.

All the displays shown here make use of the routine within a loop or loops. The repeated triangles program is based on triangles plotted between two parallel lines. Several interesting modifications are possible here; try, for example, changing the first x co-cordinate of the triangle from xc-2*y to xc-y. This will produce parallelograms between the parallel screen edges.

The spotlight display is produced by drawing a series of triangles from a single point (5,170). The base of each triangle is a horizontal line, the end points of which lie on the circumference of a shallow ellipse.

The final program, the triangle curves program, is a display of curves produced from sequences of triangles. The outer and inner curves are produced by the routines called in lines 130-140 and 180-190 respectively.

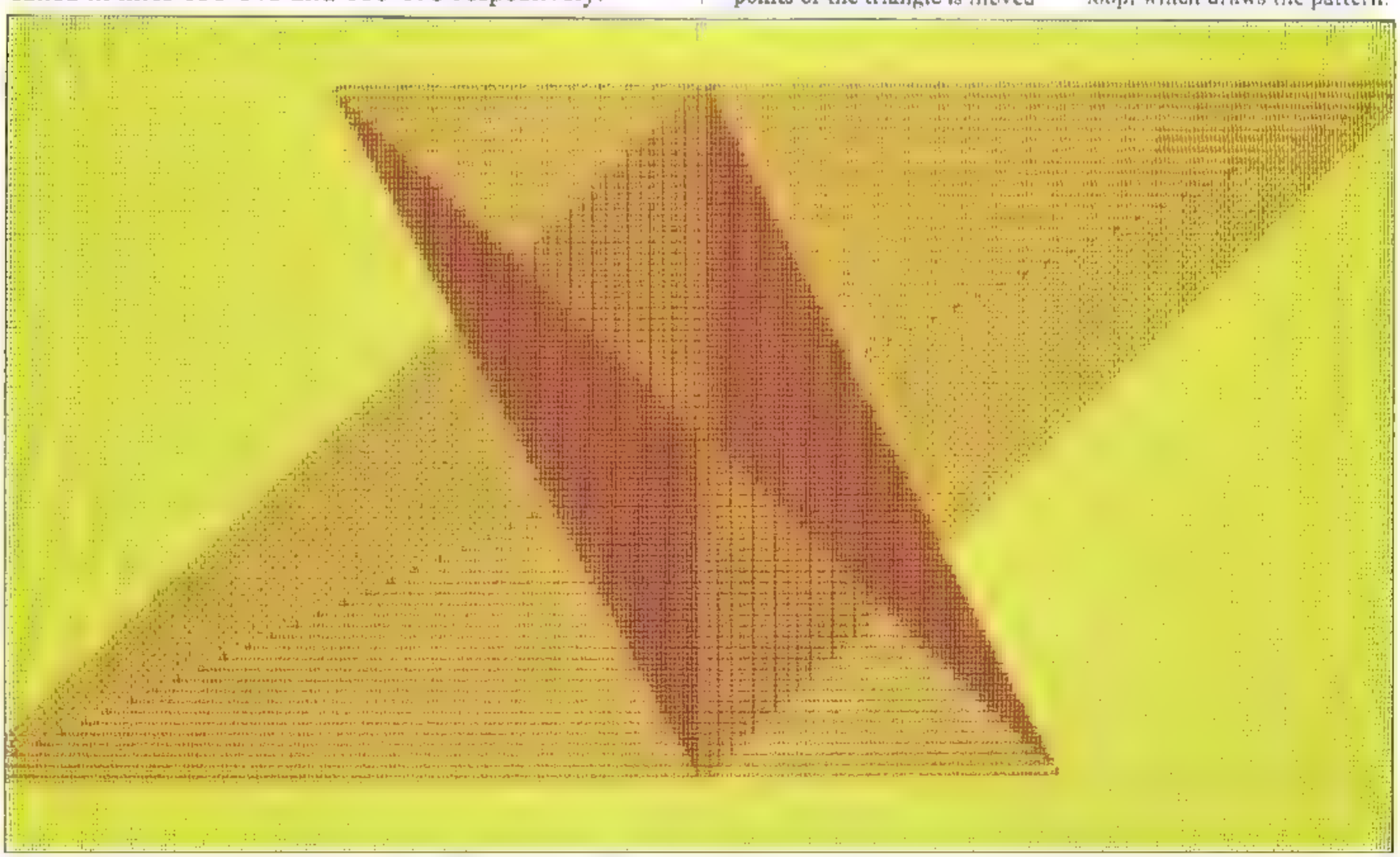
REPEATED TRIANGLES PROGRAM

```
  10 DEF FN i(x,y,p,q,r,s)=USR 5
0300
 100 BORDER 6: PAPER 6: INK 2: C
LS
 110 LET r=63: LET xc=127: LET y
c=90
 120 FOR i=7 TO 2 STEP -1
 130 FOR y=r TO -r STEP -i
 140 RANDOMIZE FN i(xc-2*y,yc-y,
xc,yc+y,xc+y,yc-y)
 150 NEXT y
 160 PAUSE 0: CLS
 170 NEXT i

0 OK, 0:1
```

REPEATED TRIANGLES PROGRAM

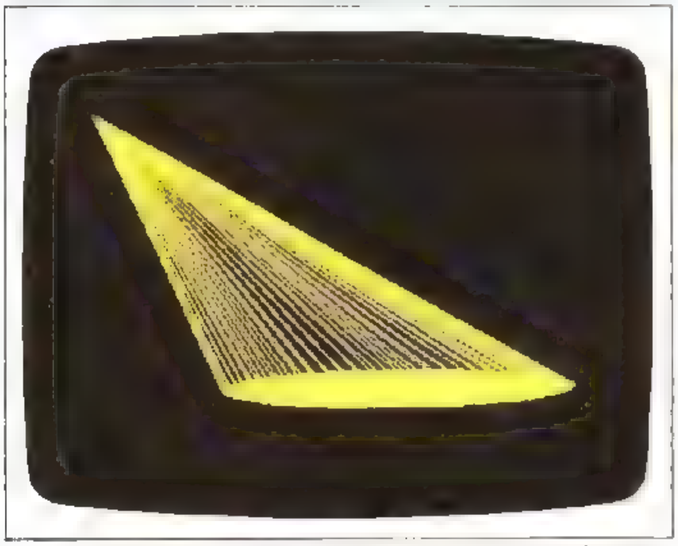
00:03 **seconds**

How the program works
The program draws a series of triangles. Each of the three points of the triangle is moved along a straight line, by changing the variable y for each successive triangle.
Line 120 begins the first loop, which sets the distance between each triangle in the display.
Line 130 sets up the second loop, which draws the pattern.



SPOTLIGHT PROGRAM

```
  10 DEF FN i(x,y,p,q,r,s)=USR 6
0300
 100 BORDER 0: PAPER [illegible]: INK 6: C
LS
 110 LET s=4: LET a=0: LET ad=s*
PI/128
 120 LET x1=162: LET y1=20
 130 FOR i=0 TO 255 STEP s
 140 LET x=x1+INT (90*SIN a)
 150 LET y=y1+INT (10*COS a)
 160 LET x2=x1+INT (90*SIN (a+PI
))
 170 RANDOMIZE FN i(x,y,x2,y,5,1
70)
 180 LET a=a+ad
 190 NEXT i

0 OK, 0:1
```



Line 160 of the curves program sets the central screen area to red using the window ink routine, FNb (which must be present in memory for the program to RUN).

TRIANGLE CURVES PROGRAM

```
  10 DEF FN b(x,y,h,v,b,c,f)=USR
 62800
  20 DEF FN i(x,y,p,q,r,s)=USR 6
0300
 100 BORDER 4: PAPER 4: INK 1
 110 CLS
 120 FOR a=0 TO 175 STEP 4
 130 RANDOMIZE FN i(a,0,0,175-a,
0,0): RANDOMIZE FN i(a,175,0,a,0
,175)
 140 RANDOMIZE FN i(255-a,175,25
5,a,255,175): RANDOMIZE FN i(255
-a,0,255,175-a,255,0)
 150 NEXT a
 160 RANDOMIZE FN b(8,4,16,14,2,
0,0)
 170 FOR a=78 TO 178 STEP 4
 180 RANDOMIZE FN i(a,38,178,a-4
0,256-a,138)
 190 RANDOMIZE FN i(256-a,138,78
,216-a,a,38)
 200 NEXT a

0 OK, 0:1
```

FNi

TRIANGLE DRAW ROUTINE

Start address 60300 **Length** 80 bytes
Other routines called Line draw routine (FNg).
What it does Draws a triangle given the pixel co-ordinates of three points.

Using the routine The routine uses absolute co-ordinates. Specifying off-screen co-ordinates produces an error message; values more than 255 pixels off the screen will probably cause the Spectrum to crash. Colours are set by the current screen INK attributes.

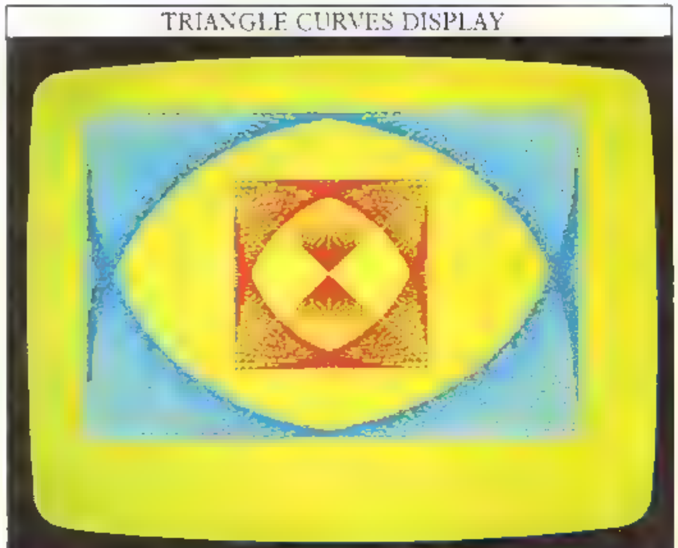
ROUTINE PARAMETERS

DEF FNi(x,y,p,q,r,s)	
x,y	specify first corner of triangle (x<256,y<176)
p,q	specify second corner of triangle (p<256,q<176)
r,s	specify third corner of triangle (r<256,s<176)

ROUTINE LISTING

```
7600 LET b=60300: LET l=75: LET
z=0: RESTORE 7610
7601 FOR i=0 TO l-1: READ a
7602 POKE (b+i),a: LET z=z+a
7603 NEXT i
7604 LET z=INT (((z/l)-INT (z/l)
)*l)
7605 READ a: IF a<>z THEN PRINT
"??": STOP

7610 DATA 42,11,92,1,4
7611 DATA 0,9,86,14,8
7612 DATA 9,94,237,83,2[illegible]8
7613 DATA 235,9,86,9,94
7614 DATA 237,83,21[illegible],235,9
7615 DATA 86,9,94,237,83
7616 DATA 212,235,42,210,235
7617 DATA 34,26,237,205,51
7618 DATA 237,237,91,208,235
7619 DATA 42,212,235,34,26

762[illegible] DATA 237,205,51,237,237
7621 DATA 91,210,23[illegible],42,208
7622 DATA 235,34,26,237,205
7623 DATA 51,237,201,40,[illegible]
7624 DATA 40,143,80,123,[illegible]
7625 DATA 68,0,0,0,0
```

TRIANGLE CURVES DISPLAY



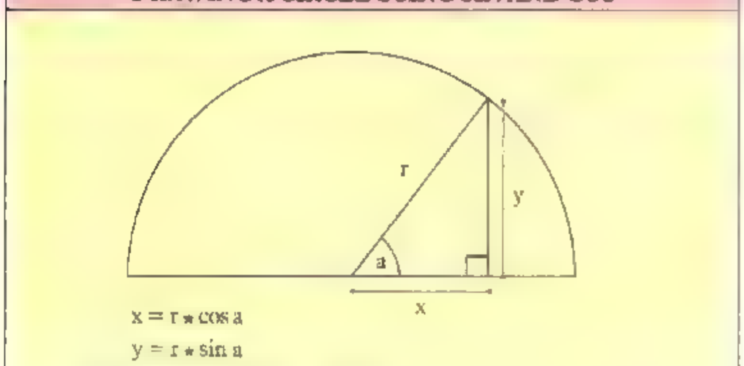
CIRCLES AND ARCS 1

Two methods are commonly used to draw circles on a computer. The first uses a combination of sines and cosines; this is the method used to draw circles in Spectrum BASIC. The sine/cosine method is derived from the fact that, for a right-angled triangle, the length of the horizontal and vertical sides can be calculated from the size of one angle and the length of the third side. If a right-angled triangle is formed between the centre of a circle and any point on the circumference, as shown in the diagram below, then the length of the sides is given by

$$x = r * \cos(a)$$
$$y = r * \sin(a)$$

You can see a typical example of circles plotted in Spectrum BASIC in the circle program on this page. The command requires the centre-point and radius to be specified (x,y and r).

DRAWING A CIRCLE USING SIN AND COS

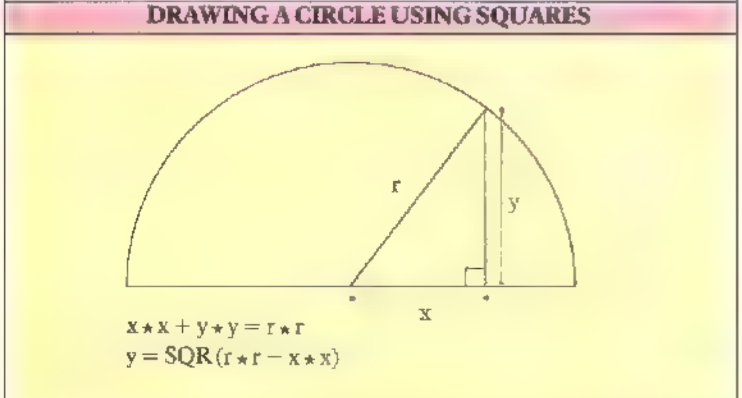


The second method is faster in operation but requires more memory to implement. This method, based on squares, forms the basis of the machine-code routine given here. It is derived from the equation

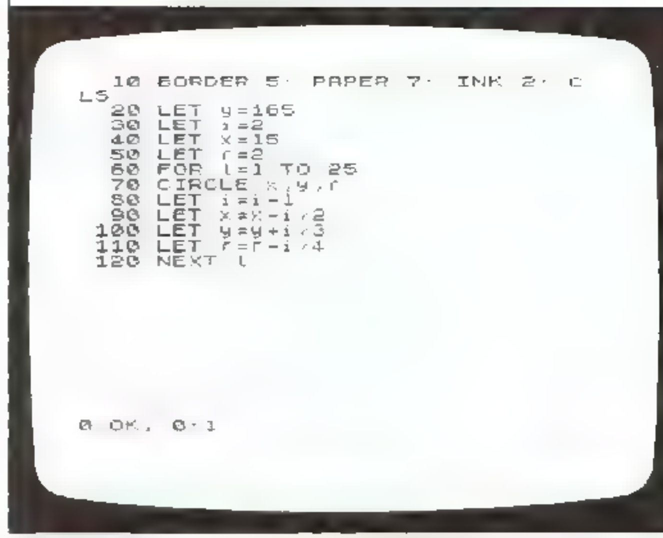
$$x^2 + y^2 = r^2$$

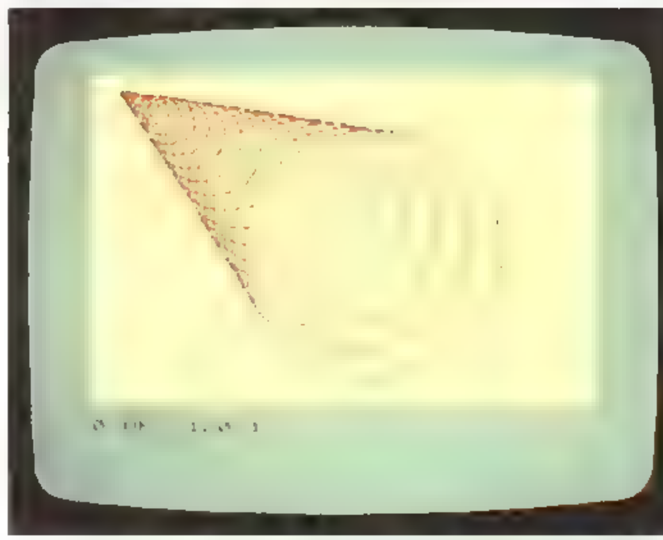
This, of course, is Pythagoras' theorem, which gives the relation between the sides of a right-angled triangle, as shown in the squares method diagram below.

DRAWING A CIRCLE USING SQUARES



BASIC CIRCLES PROGRAM





This method is more complicated than the sine/cosine method, since it must calculate square root values each time a circle is drawn. To use it most effectively, first calculate a list of square roots and then store this list in memory — as done by the routine at address 59600. The list of square roots can then be "consulted" by the main routine. Using stored square roots makes this routine much faster — and more accurate — than using the BASIC CIRCLE command.

Using the routines

Together with the BASIC square loader program, the routines given here do the work of calculating points for circles to be drawn. After keying in these routines you will not yet be able to produce anything on the screen, because the routines do not in themselves draw any curves; for this you must also key in one of the curve routines in the following pages.

MASTER CURVE ROUTINES

Start addresses 59600 and 59000
Length 60 and 525 bytes
Other routines called Point-plot routine (FNf).
What they do Carry out the calculations for the arc, sector and segment routines.

Using the routines The following BASIC program must be keyed in and RUN before using the machine-code routines given here:

```
10 LET j = 59700
20 FOR i = 0 TO 255
30 LET p = i * i : LET h = INT (p / 256)
40 LET l = p—256 * h
50 POKE j,l : POKE j +1,h
60 LET j = j + 2
70 NEXT i
```

This program POKEs the squares of numbers from 0 to 255 into memory. Each square is stored in two bytes, since numbers larger than 16 squared will not fit into a single byte, which has a maximum value of 255. Having keyed in this routine, SAVE the area of memory containing the squares by using the command SAVE "title" CODE 59700,600. These 600 bytes are also used as workspace by the circle routines.
The routine at 59600 calculates square roots and stores them in memory. The longer routine, starting at address 59000, calculates points on the circumference of a circle using these square roots.

ROUTINE LISTING

```
7650 LET b=59600. LET l=55: LET
z=0: RESTORE 7660
7651 FOR i=0 TO l-1. READ a
7652 POKE (b+i),a: LET z=z+a
7653 NEXT i
7654 LET z=INT (((z/l)-INT (z/l)
)*l)
7655 READ a: IF a<>z THEN PRINT
"??": STOP

7660 DATA 62,0,186,32,4
7661 DATA 187,32,1,201,1
7662 DATA 52,233,10,111,3
7663 DATA 10,103,167,237,82
7664 DATA 242,234,232,3,24
7665 DATA 242,17,202,22,96
7666 DATA 105,25,124,167,31
7667 DATA 125,31,201,33,52
7668 DATA 233,22,0,7,48
7669 DATA 1,20,95,25,94
7670 DATA 35,86,201,0,0
7671 DATA 3,0,0,0,0
```

ROUTINE LISTING

```
7700 LET b=59000: LET l=520: LET
 z=0: RESTORE 7710
7701 FOR i=0 TO l-1: READ a
7702 POKE (b+i),a: LET z=z+a
7703 NEXT i
7704 LET z=INT (((z/l)-INT (z/l)
)*l)
7705 READ a: IF a<>z THEN PRINT
"??": STOP

7710 DATA 62,1,50,104,232
7711 DATA 58,112,232,205,246
7712 DATA 232,237,83,117,232
7713 DATA 167,203,26,203,27
7714 DATA 205,208,232,50,119
7715 DATA 232,58,113,232,205
7716 DATA 11,232,50,120,232
7717 DATA 58,114,232,205,11
7718 DATA 232,50,121,232,58
7719 DATA 113,232,205,19,232
7720 DATA 50,115,232,58,114
7721 DATA 232,205,19,232,50
7722 DATA 116,232,71,58,115
7723 DATA 232,144,200,58,115
7724 DATA 232,50,122,232,58
7725 DATA 120,232,71,58,121
7726 DATA 232,144,32,34,50
7727 DATA 50,123,232,58,121
7728 DATA 232,230,1,40,9
7729 DATA 58,116,232,50,124

7730 DATA 232,195,7,231,58
7731 DATA 116,232,71,58,119
7732 DATA 232,144,50,124,232
7733 DATA 195,7,231,60,50
7734 DATA 123,232,58,120,232
7735 DATA 230,1,32,8,62
7736 DATA 0,50,124,232,195
7737 DATA 7,231,58,119,232
7738 DATA 50,124,232,58,120
7739 DATA 232,71,33,105,231

7740 DATA 17,18,0,25,16
7741 DATA 253,34,125,232,58
7742 DATA 120,232,230,1,40
7743 DATA 31,58,122,232,50
7744 DATA 122,232,205,63,232
7745 DATA 42,125,232,233,205
7746 DATA 81,232,58,122,232
7747 DATA 33,124,232,190,40
7748 DATA 43,60,250,98,231
7749 DATA 24,228,58,122,232

7750 DATA 71,58,119,232,167
7751 DATA 144,50,122,232,205
7752 DATA 63,232,42,125,232
7753 DATA 233,205,81,232,58
7754 DATA 122,232,33,124,232
7755 DATA 190,40,6,61,250
7756 DATA 98,231,24,228,58
7757 DATA 120,232,60,50,120
7758 DATA 232,62,0,50,122
7759 DATA 232,58,123,232,61

7760 DATA 200,254,1,202,205
7761 DATA 230,195,239,230,87
7762 DATA 58,122,232,95,58
7763 DATA 111,232,130,87,58
7764 DATA 110,232,131,95,195
7765 DATA 43,231,95,58,122
7766 DATA 232,87,58,111,232
7767 DATA 130,87,58,110,232
7768 DATA 131,95,195,80,231
7769 DATA 95,58,122,232,87

7770 DATA 58,111,232,146,87
7771 DATA 58,110,232,131,95
7772 DATA 195,43,231,87,58
7773 DATA 122,232,95,58,111
7774 DATA 232,146,87,58,110
7775 DATA 232,131,95,195,80
7776 DATA 231,87,58,122,232
7777 DATA 95,58,111,232,146
7778 DATA 87,58,110,232,147
7779 DATA 95,195,43,231,95

7780 DATA 58,122,232,87,58
7781 DATA 111,232,146,87,58
7782 DATA 110,232,147,95,195
7783 DATA 80,231,95,58,122
7784 DATA 232,87,58,111,232
7785 DATA 130,87,58,110,232
7786 DATA 147,95,195,43,231
7787 DATA 87,58,122,232,95
7788 DATA 58,111,232,130,87
7789 DATA 58,110,232,147,95

7790 DATA 195,80,231,6,5
7791 DATA 203,63,16,252,60
7792 DATA 201,230,31,254,0
7793 DATA 200,71,175,79,237
7794 DATA 91,117,232,33,0
7795 DATA 0,25,48,4,12
7796 DATA 32,1,60,16,247
7797 DATA 93,84,111,6,6
7798 DATA 167,203,29,203,25
7799 DATA 203,26,203,27,16

7800 DATA 245,205,208,232,201
7801 DATA 58,122,232,205,246
7802 DATA 232,42,117,232,167
7803 DATA 237,82,93,84,205
7804 DATA 208,232,201,58,104
7805 DATA 232,254,0,40,8
7806 DATA 175,50,104,232,237
7807 DATA 83,106,232,237,83
7808 DATA 108,232,205,72,240
7809 DATA 201,0,0,95,165
7810 DATA 52,130,80,140,30
7811 DATA 20,170,16,11,132
7812 DATA 3,21,7,6,0
7813 DATA 1,10,213,231,0
7914 DATA 234,0,0,0,0
```

CIRCLES AND ARCS 2

With the arc routine, FNj, given on this page, you will be able to draw arcs more quickly and with more flexibility than with the Spectrum DRAW command. The routine can be used to draw either arcs or complete circles by varying the final two parameters.

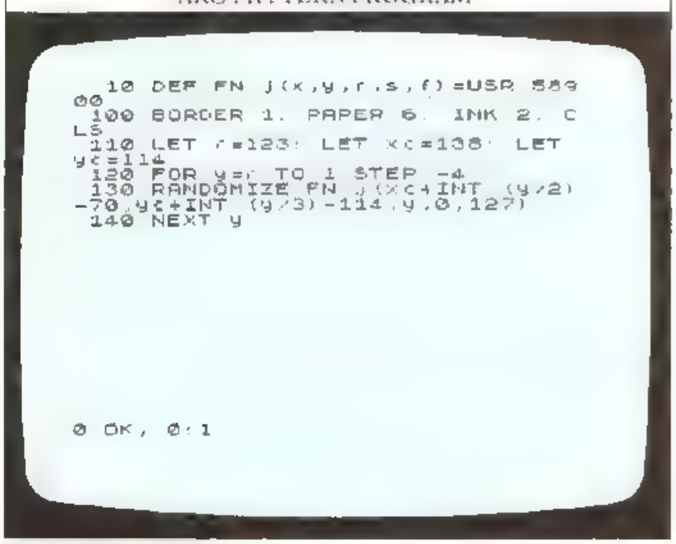
To produce any circle-based program on this page, you must first key in the machine-code routines and the BASIC squares program on page 29, as well as the routine given on this page, since the arc routine calls all these routines.

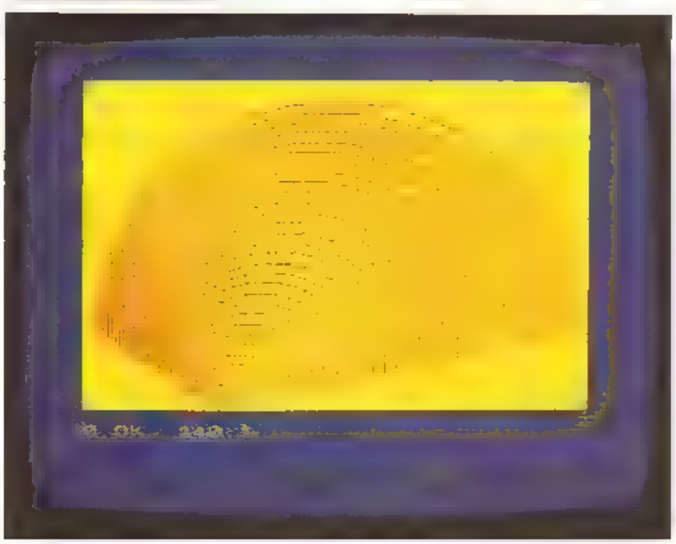
Starting and finishing the arc

The only complication of the arc routine given here is in specifying how much of the circle is to be drawn. The start and finish parameters can have values from 0 to 255, instead of 0 to 360. This is because the parameters are stored in a single byte in memory, and one byte can have only 256 values (that is, from 0 to 255). This means that s,f values of 0,255 will draw a complete circle, values 0,127 will give a semicircle, and so on.

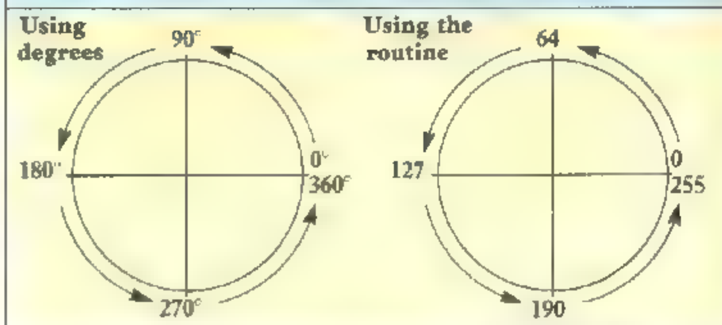
Since the routine begins drawing from a position horizontally to the right of the selected centre point, s,f values of 0,64 will produce a quarter circle from the right of the centre to a point vertically above it. The diagram below shows how the start and finish parameters correspond to the more usual degrees.

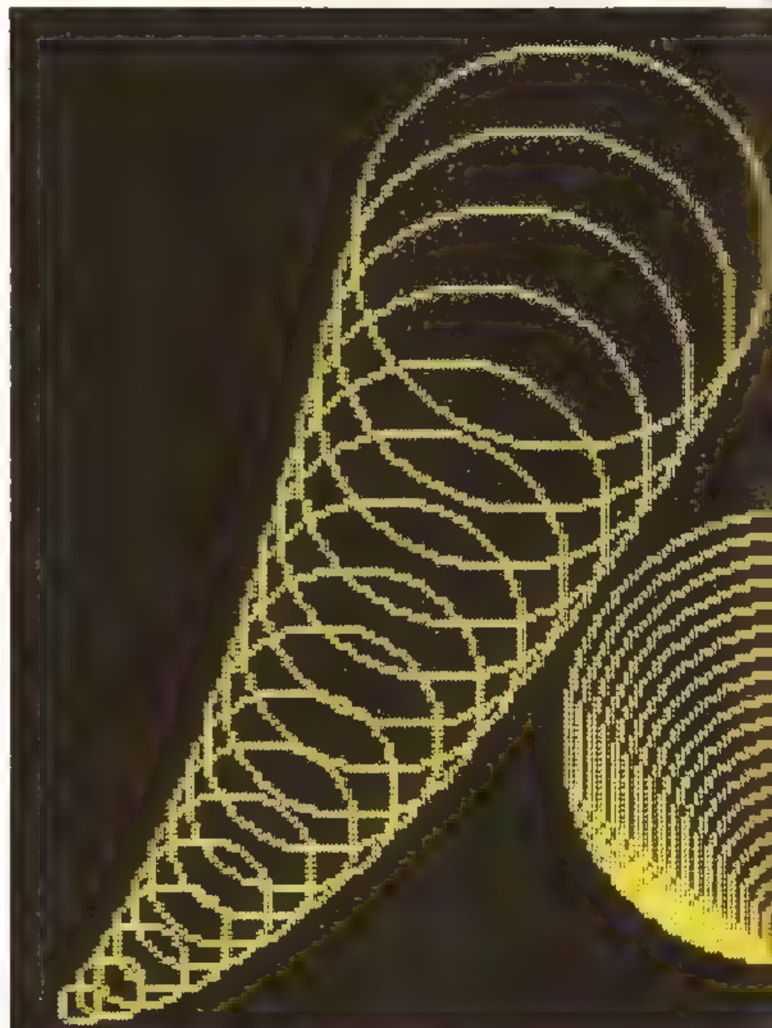
ARC PATTERN PROGRAM





CALCULATING POINTS ON A CIRCLE



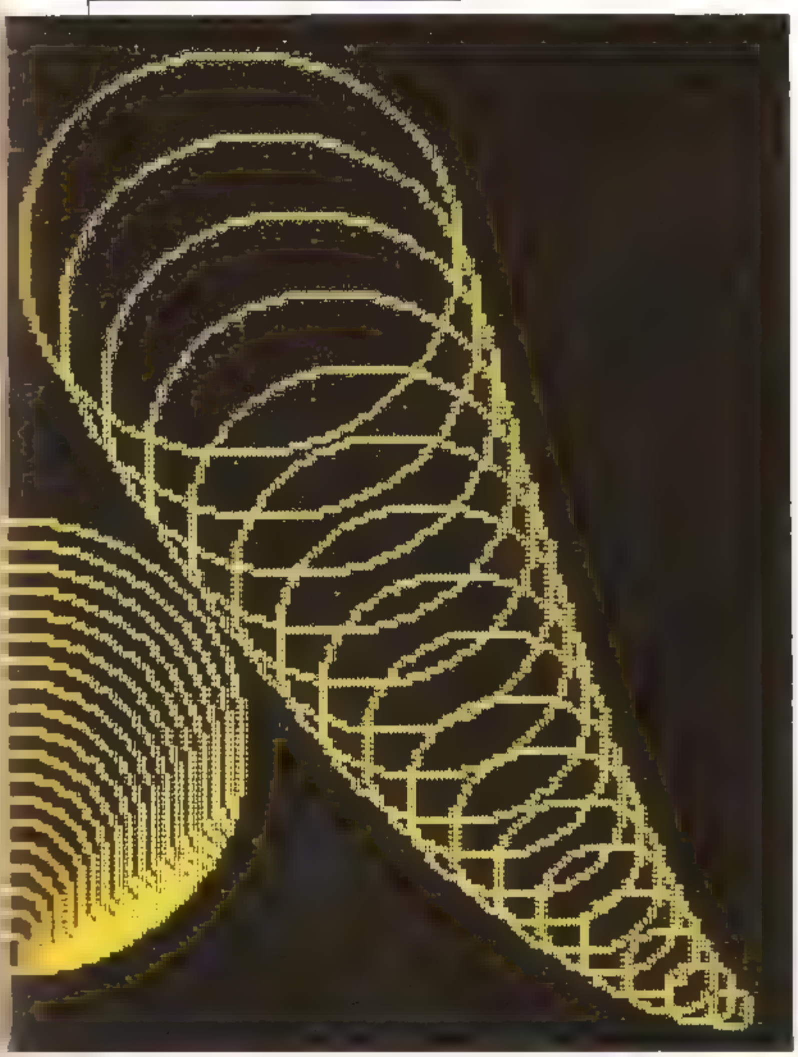


FNj

ARC ROUTINE

Start address 58900 **Length** 45 bytes
Other routines called Master curve routines.
What it does Draws an arc or circle at a specified radius from a centre point.

Using the routine The table on the facing page shows how the s and f parameters specify the length of the arc. A difference of 255 will produce a complete circle, 127 a semicircle, and so on. The numbers themselves define the angle from the centre of the circle at which the arc starts and finishes. The arc is drawn from a position due east of the centre of the circle, so that an s or f value of 1 is to the right of the centre point, a value of 128 to the left, and a value of 192 directly beneath the centre of the circle.
Unlike CIRCLE in Spectrum BASIC, you can draw curves with this routine which go some distance off the top or bottom of the screen without an error message appearing.
Because of the way the screen memory works, there are several screen positions where a curve cannot be drawn using this routine. If you find the routine does not work in any position, move the centre point one pixel in any direction.



ROUTINE PARAMETERS

DEF FNj(x,y,r,s,f)	
x,y	specify the centre point from which the arc is drawn (x<256,y<176)
r	specifies the radius of the arc (r<256)
s,f	specify the length of the arc(s<f, s<256,f<256)

ROUTINE LISTING

```
7850 LET b=58900: LET l=40: LET
z=0: RESTORE 7860
7851 FOR i=0 TO l-1: READ a
7852 POKE (b+i),a: LET z=z+a
7853 NEXT i
7854 LET z=INT (((z/l)-INT (z/l)
)*l)
7855 READ a: IF a<>z THEN PRINT
"??": STOP

7860 DATA 42,11,92,1,4
7861 DATA 0,9,86,14,8
7862 DATA 9,94,237,83,110
7863 DATA 232,9,126,50,112
7864 DATA 232,9,126,50,113
7865 DATA 232,9,126,50,114
7866 DATA 232,71,58,113,232
7867 DATA 176,200,195,120,230
7868 DATA 17,0,0,0,0
```

CONES PROGRAM

```
  10 DEF FN j(x,y,r,s,f)=USR 589
00
 100 BORDER 0: PAPER 0: INK 6
 110 CLS
 120 FOR i=1 TO 18
 130 RANDOMIZE FN j(254-i*5,INT
(i↑1.7),2*i,0,255)
 140 NEXT i
 150 FOR i=1 TO 18
 160 RANDOMIZE FN j(i*5-4,INT (i
↑1.7),2*i,0,255)
 170 NEXT i
 180 FOR i=1 TO 20
 190 RANDOMIZE FN j(128,10+i*2,2
*i,0,255)
 200 NEXT i



0 OK, 0:1
```

The cones program produces patterns by varying the x and y co-ordinates of the centre of a circle each time it is drawn. In the left- and right-hand patterns, the x coordinate is a function of the variable i, while the y coordinate is given by i raised to the power of 1.7. As a result, the circles appear on a curve. In the third loop, only the y co-ordinate is varied, so that the sequence of circles rises vertically.

CONES PROGRAM

00:11 **seconds**

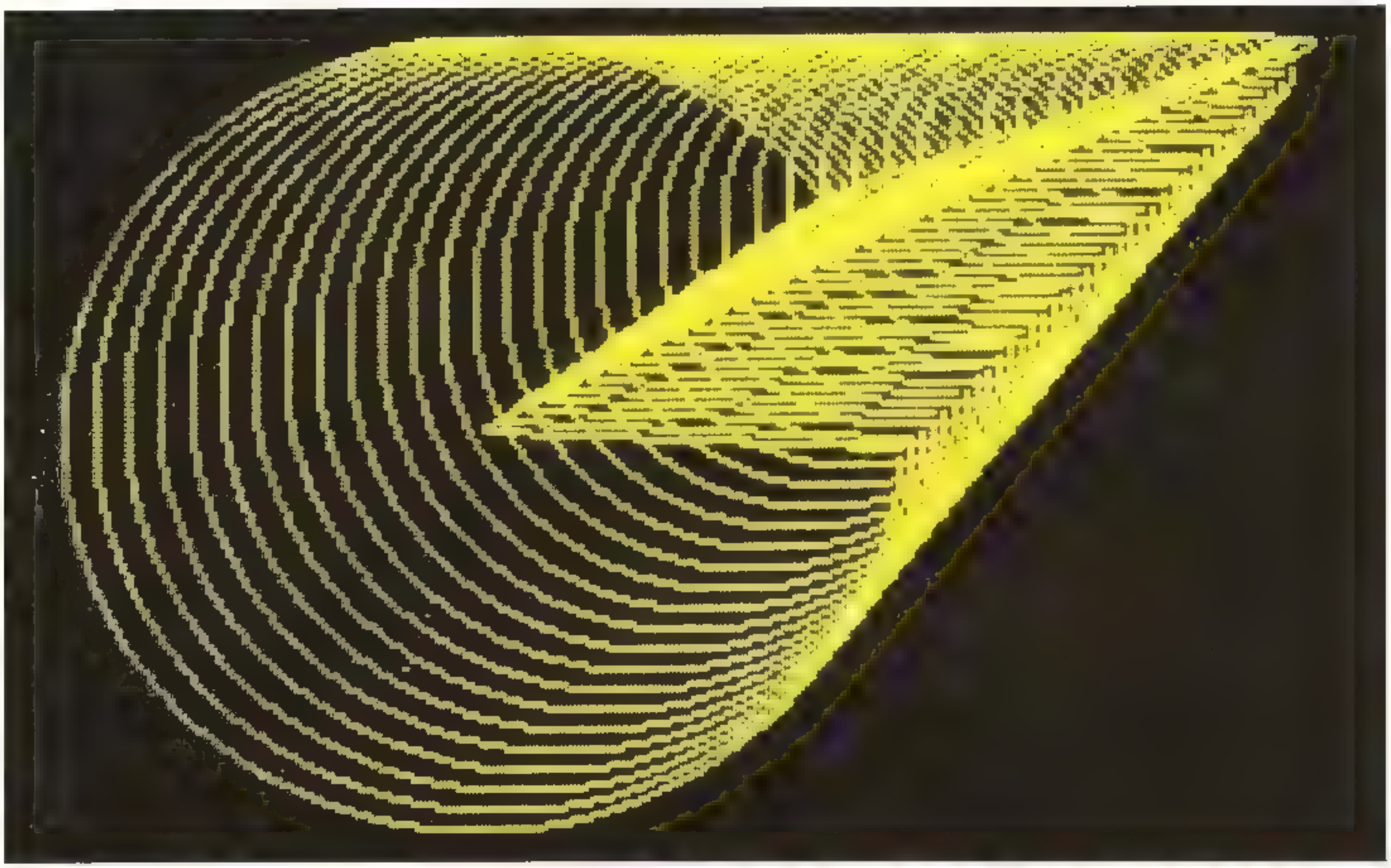
How the program works
Three circle patterns are drawn, using the circle routine within a loop.

Lines 120-140 draw the left-hand circles.
Lines 150-170 repeat the above loop, reversing the x,y co-ordinates.
Lines 180-200 draw the centre circles.

SECTORS AND SEGMENTS

The two routines on this page are useful supplements to the circle routine introduced on pages 30-31. Sectors are constructed by drawing an arc and then joining the end points to the centre from which the arc is drawn. A segment differs from a sector in that the ends of a segment are joined to each other, rather than to a centre point. The advantage of the routine is that they enable you to join the ends of an arc together without having to work out the co-ordinates of those points.

Both the sector and the segment routines call the master circle routine (page 29), the arc routine, FNj (page 31) and the line draw routine, FNn (page 21). This means that the sector and segment routines will not work unless these other routines are present in memory.



SECTOR PROGRAM

```
  10 DEF FN k(x,y,r,s,f)=USR 588
00
 100 BORDER 0: PAPER 0: INK 4: C
LS
 110 LET r=80: LET xc=250: LET y
c=170
 120 FOR i=3 TO 2 STEP -1
 130 FOR y=r TO 1 STEP -i
 140 RANDOMIZE FN k(xc-y*2,yc-IN
T (y),y,20,250)
 150 NEXT y
 160 PAUSE 0: CLS
 170 NEXT i

0 OK, 0:1
```

The sector program on this page creates the illusion of a third dimension by repeatedly drawing smaller and smaller sectors while at the same time moving the centre point upwards and to the right. Try varying i in line 120 to see a different number of sectors displayed.

The segment program repeats a pattern of segments (drawn from a single centre point in lines 140-150) three times across the screen. The number of patterns can be increased by varying the step size of x in line 120.

SECTOR PROGRAM

00:11 **seconds**

How the program works A sector of a circle is drawn repeatedly with decreasing radius.

Line 120 sets up a loop to vary i, the number of sectors drawn in one display.

Line 130 sets up a loop to vary y, used to calculate the centre point and the radius.

Line 140 draws a single sector.

Line 160 waits for a key to be pressed before drawing the display again.

FNk

SECTOR ROUTINE

Start Address 58800 **Length** 45 bytes
Other routines called Arc and line-draw routines (FNj, FNg).
What it does Draws an arc of specified radius, and joins each end to the centre point.

Using the routine The sector is drawn anti-clockwise from a point to the right of the centre. When the ends of the arc are joined to the centre, the result is a wedge shape if the difference between s and f is less than 127, or a cut pie shape if the difference is greater than 128. Sectors plotted off the screen to left or right may reappear rather unpredictably elsewhere on the screen, so it is best to keep within the parameter limits given below.

ROUTINE PARAMETERS

DEF FNk(x,y,r,s,f)	
x,y	specify the centre point from which the arc is to be drawn (x<256,y<176)
r	specifies the radius of the arc (r<256)
s,f	specify the length of the arc (s<f, s<256, f<256)

ROUTINE LISTING

```
7900 LET b=5880■: LET l=40: LET
z=0: RESTORE 7910
7901 FOR i=0 TO l-1: READ a
7902 POKE (b+i),a: LET z=z+a
7903 NEXT i
7904 LET z=INT (((z/l)-INT (z/l)
)*l)
7905 READ a: IF a<>z THEN PRINT
"??": STOP

7910 DATA 205,20,230,237,91
7911 DATA 106,232,42,110,232
7912 DATA 229,0,34,26,237
7913 DATA 205,51,237,237,91
7914 DATA 108,232,225,34,26
7915 DATA 237,205,51,237,237
7916 DATA 91,110,232,205,72
7917 DATA 240,201,0,0,0
7918 DATA 35,0,0,0,0
```

FNl

SEGMENT ROUTINE

Start address 58700 **Length** 30 bytes
Other routines called Arc and line-draw routines (FNj, FNg).
What it does Draws an arc of specified radius from a centre point, and joins the ends together.

Using the routine This routine works in the same way and with the same restrictions as the sector routine, except that in this case the ends of the arc are joined together, rather than to the centre.
Notice that, like the previous routine, you may get problems trying to connect the ends of the arc together, if either of the end points (and especially if both of them) are off the screen. As before, segments plotted off the edge of the screen to left or right will have unpredictable results: they may reappear on the other side, or cause the Spectrum to crash.

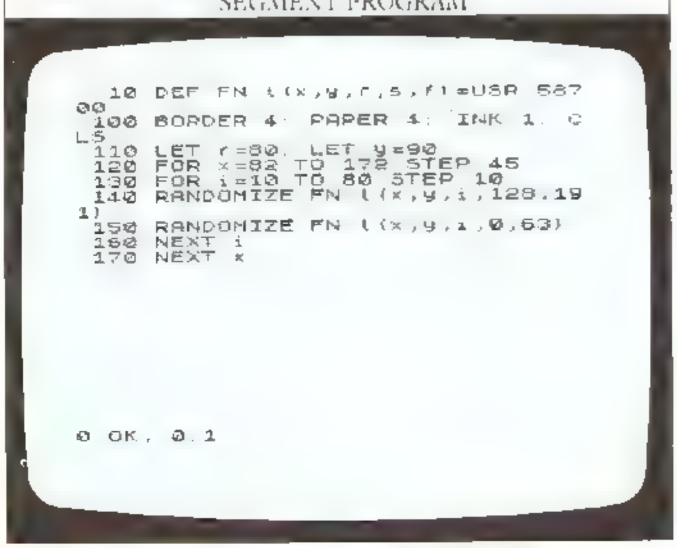
ROUTINE PARAMETERS

DEF FNl (x,y,r,s,f)	
x,y	specify the centre point from which the arc is to be drawn (x<256,y<176)
r	specifies the radius of the arc (r<256)
s,f	specify the length of the arc (s<f, s<256, f<256)

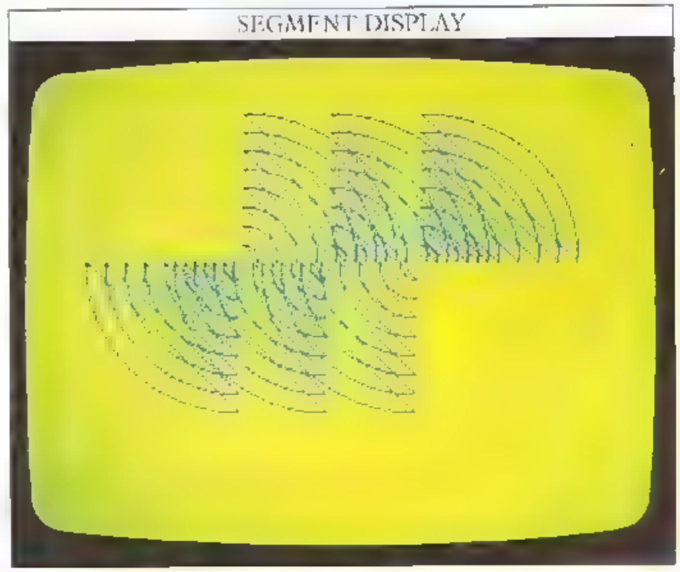
ROUTINE LISTING

```
7950 LET b=58700: LET l=25: LET
z=0: RESTORE 7960
7951 FOR i=0 TO l-1: READ a
7952 POKE (b+i),a: LET z=z+a
7953 NEXT i
7954 LET z=INT (((z/l)-INT (z/l)
)*l)
7955 READ a: IF a<>z THEN PRINT
"??": STOP

7960 DATA 205,20,230,237,91
7961 DATA 106,232,0,42,108
7962 DATA 232,0,0,34,26
7963 DATA 237,205,51,237,0
7966 DATA 201,0,0,0,0
7967 DATA 18,0,0,0,0
```

SEGMENT PROGRAM



SEGMENT DISPLAY



FILLING SHAPES 1

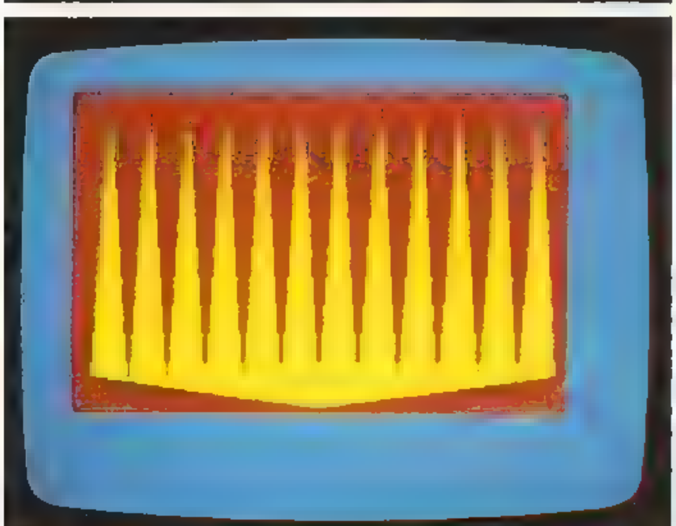
The fill routine given here, FNm, enables you to fill any enclosed shape no matter how irregular. The routine works by looking at the pixels adjacent to the specifed start point. If a pixel INK attribute is set, the routine does not change it, and does not look at pixels adjacent to this one; otherwise, the routine sets the INK attribute to the current INK colour and moves to the next adjacent pixels.

This method is known as the flood or grass-fire method, since, as you can see from its characteristic diamond shape, the INK spreads outwards until it reaches a "trench", which stops it from spreading further. Any shape which is not completely enclosed, even if only by a single pixel, will "leak" when filled.

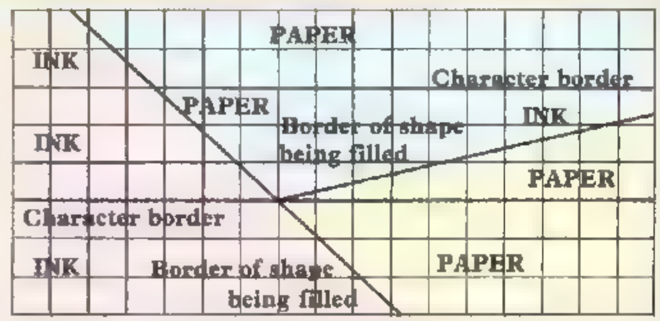
Colouring irregular shapes

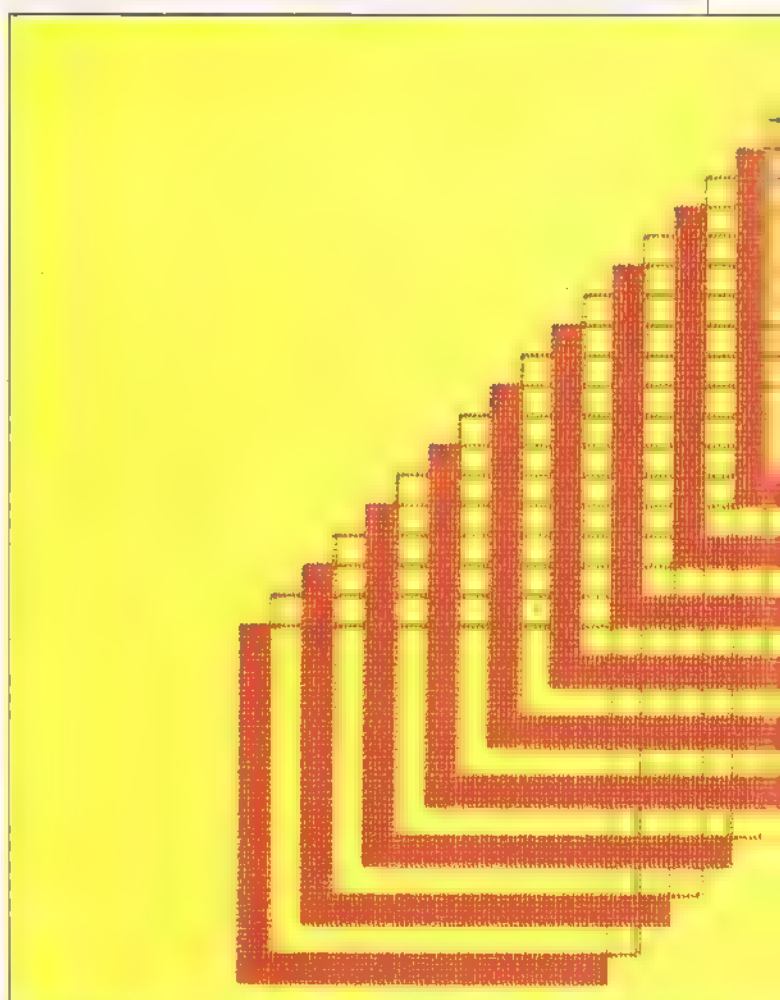
Since the Spectrum can only have one INK and one PAPER colour in each character block, you may have problems when there are more than two colours on the screen, and you call the routine to fill irregular shapes. If, for example, the shape has diagonal edges, you will see a jagged effect corresponding to character borders, instead of a straight line when the shape is filled. The diagram below shows how a combination of INK and PAPER colours can be used to overcome this problem.

FILL PROGRAM



FILLING SHAPES AT CHARACTER BORDERS





BOX FILL PROGRAM

```
  10 DEF FN h(x,y,h,v)=USR 60400
  20 DEF FN m(x,y)=USR 57700
 100 BORDER 2: INK 2: PAPER 6: C
LS
 110 LET x=140
 120 FOR j=110 TO 10 STEP -5
 130 RANDOMIZE FN h(x,j,60,60)
 140 IF x/10=INT (x/10) THEN RAN
DOMIZE FN m(x+1,j+1)
 150 LET x=x-5
 160 NEXT j



0 OK, 0:1
```

BOX FILL PROGRAM

00:05 seconds

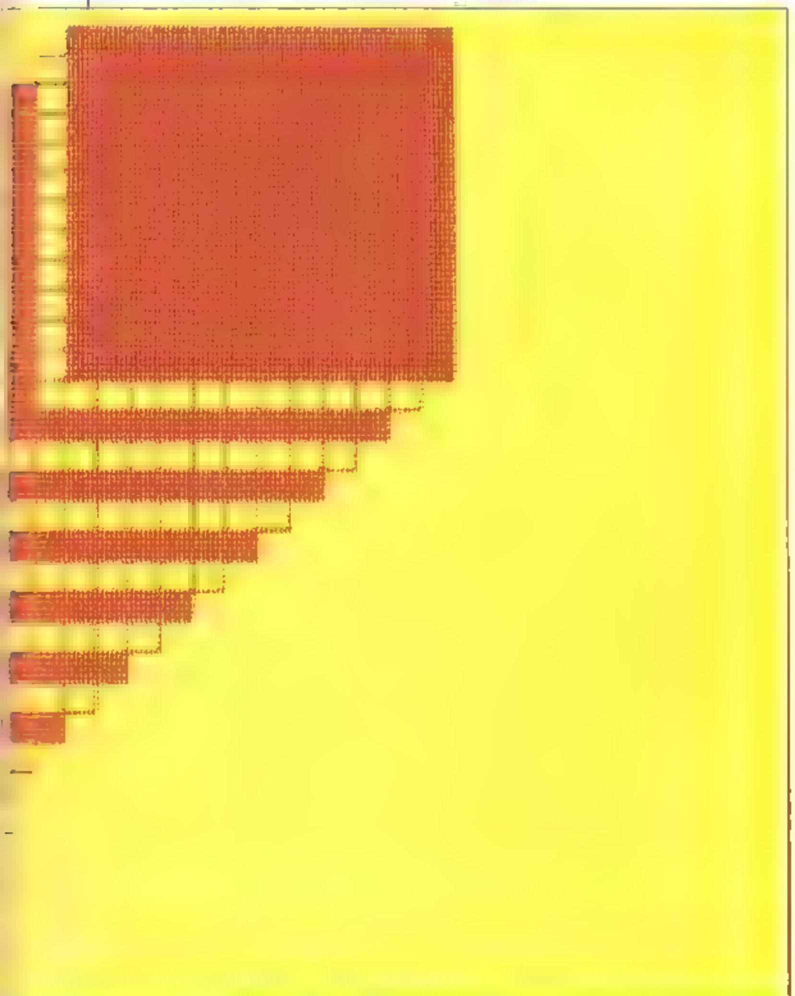
How the program works

Boxes are drawn in a loop, and filled alternately.

Line 120 sets up a loop.

Line 140 fills a box if variable x is exactly divisible by 10.

Line 150 reduces the value of x by 5.



FNm

FILL ROUTINE

Start address 57700 **Length** 195 bytes
Other routines called Line-draw routine (FNg).
What it does Fills in an area bounded by a solid line of INK, in the current INK colour.

Using the routine This routine fills in an area up to the edge of any shape enclosed by an INK line, or to the screen border. Remember that if there is even a single pixel of PAPER colour at the border, then the INK with which you are filling will leak out, and you may fill the entire screen. Notice also that a "wraparound" effect occurs when filling to left and right of the screen, which means that when the routine reaches the left-hand edge of the screen, it starts filling from the right-hand edge inwards, and the same will happen when the routine reaches the right-hand screen edge.
If some of the attributes for character squares within the area to be filled differ from each other (as will happen, for example, if you change some of the attributes using the window ink routine) then the area will be filled with these colours, rather than in a single colour.

ROUTINE PARAMETERS

DEF FNm(x,y)	
x,y	pixel co-ordinates of the point at which to start filling (x<256, y<176)

ROUTINE LISTING

```
8000 LET b=57700: LET l=190: LET
 z=0: RESTORE 8010
8001 FOR i=0 TO l-1: READ a
8002 POKE (b+i),a: LET z=z+a
8003 NEXT i
8004 LET z=INT (((z/l)-INT (z/l)
)*l)
8005 READ a: IF a<>z THEN PRINT
"??": STOP

8010 DATA 42,11,92,1,4
8011 DATA 0,9,86,14,8
8012 DATA 9,94,237,83,44
8013 DATA 226,237,83,42,225
8014 DATA 33.44,226,229,35
8015 DATA 35,34,40,226,225
8016 DATA 34,38,226,42,38
8017 DATA 226,94,35,86,21
8018 DATA 205,207,225,42,38
8019 DATA 226,94,28,35,86

8020 DATA 205,207.225,42,38
8021 DATA 225.94,35.86,20
8022 DATA 205,207,225,42,38
8023 DATA 226,94,29,35,86
8024 DATA 205,207,225,42,38
8025 DATA 226,35,35,229,1
8026 DATA 76,229,167,237,66
8027 DATA 32,5,225,33,44
8028 DATA 226,229.225,34,38
8029 DATA 226,237,75,40,226

8030 DATA 167,237,66,200,195
8031 DATA 133,22[illegible],237,83,42
8032 DATA 226,62,175,147,216
8033 DATA 95,167.31,55,31
8034 DATA 167,31,171,230,248
8035 DATA 171,103,122,7,7
8036 DATA 7,171,230,199,171
8037 DATA 7,7,111,122,230
8038 DATA 7,71.4,62,254
8039 DATA 15,16,253,6,255

8040 DATA 168,71,126,160,192
8041 DATA 126,176,119,42,40
8042 DATA 225.237.91,42,226
8043 DATA 115,35,114,35,229
8044 DATA 1,76,229,167,237
8045 DATA 66,32,5,225,33
8046 DATA 44,226,229,225,34
8047 DATA 40,226,201,193,195
8048 DATA 57,0,0,0,0
```

FILLING SHAPES 2

The fill routine really comes into its own when it is given highly irregular shapes to fill. Not only does it cope with these shapes with ease, it also fills them very quickly. The two programs on this page give an idea of the routine's capabilities.

The only complicated detail in each program is the calculation of the point from which the fill routine is to start. Each program has to calculate this point on each pass of the loop. To ensure that whole numbers are passed to the routine, the formula for the co-ordinates of the point is placed in brackets and an INT statement placed in front of it.

SQUARES AND CIRCLES PROGRAM

00:09 **seconds**

How the program works A series of boxes and circles of increasing size is drawn, and the fill routine called inside areas at which the boxes and circles intersect.

Lines 10-30 define the routines.

Line 120 sets a centre point for the display (x1,y1).

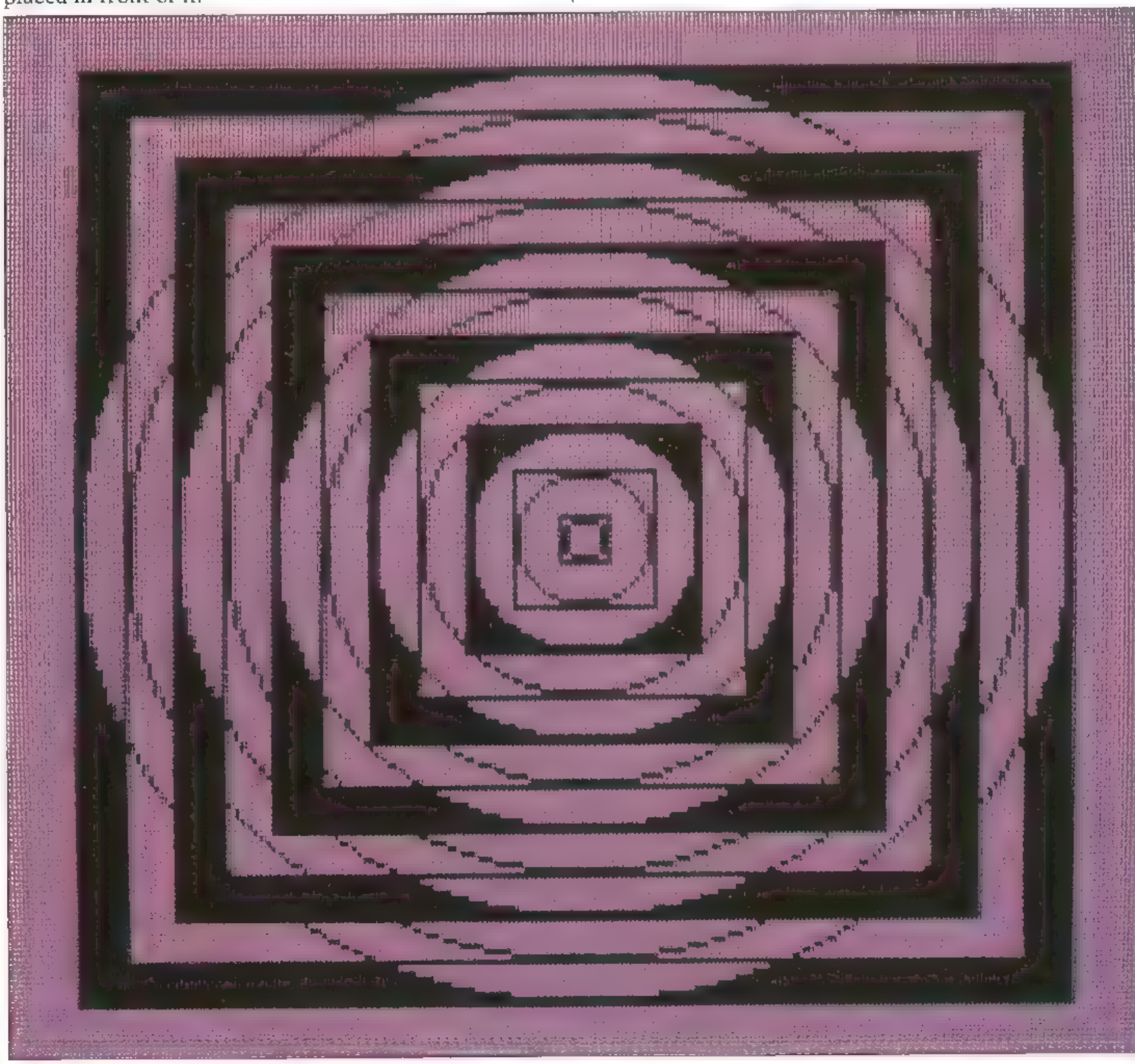
Line 130 starts a loop to draw the boxes and circles.

Line 140 draws a box based on an increment from the centre point.

Line 150 draws a circle.

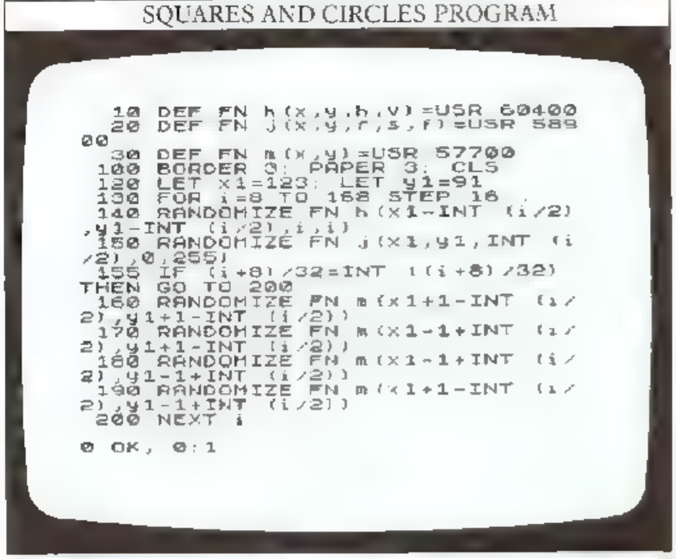
Line 155 sets a test which calls the fill routine on alternate passes of the loop only.

Lines 160-190 fill four corners of the display.

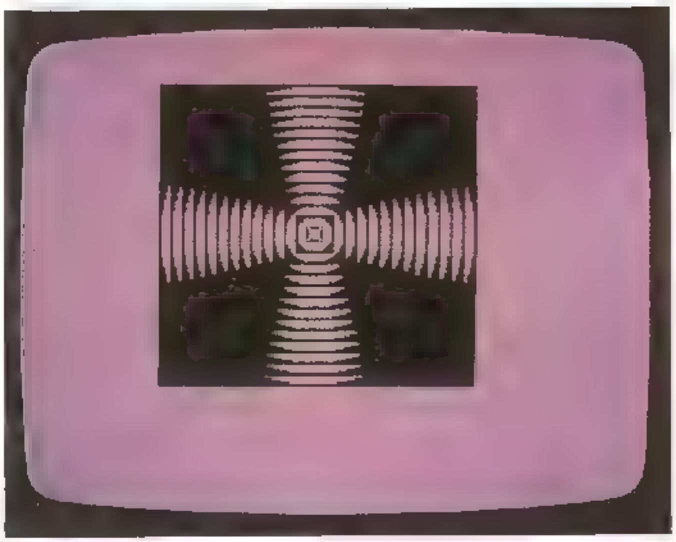


The squares and circles program fills in the intersections between a series of boxes and circles. The program is interesting for the different final displays which can be obtained by changing the values of a few parameters, caused by different shapes created each time the boxes and circles are drawn. The boxes and circles are drawn in a loop at lines 140-150, and the intersections filled at lines 160-190.

SQUARES AND CIRCLES PROGRAM



```
  10 DEF FN h(x,y,h,v)=USR 60400
  20 DEF FN j(x,y,r,s,f)=USR 589
00
  30 DEF FN m(x,y)=USR 57700
 100 BORDER 3: PAPER 3: CLS
 120 LET x1=123: LET y1=91
 130 FOR i=8 TO 168 STEP 16
 140 RANDOMIZE FN h(x1-INT (i/2)
,y1-INT (i/2),i,i)
 150 RANDOMIZE FN j(x1,y1,INT (i
/2),0,255)
 155 IF (i+8)/32=INT ((i+8)/32)
THEN GO TO 200
 160 RANDOMIZE FN m(x1+1-INT (i/
2),y1+1-INT (i/2))
 170 RANDOMIZE FN m(x1-1+INT (i/
2),y1+1-INT (i/2))
 180 RANDOMIZE FN m(x1-1+INT (i/
2),y1-1+INT (i/2))
 190 RANDOMIZE FN m(x1+1-INT (i/
2),y1-1+INT (i/2))
 200 NEXT i

0 OK, 0:1
```



The spiral program

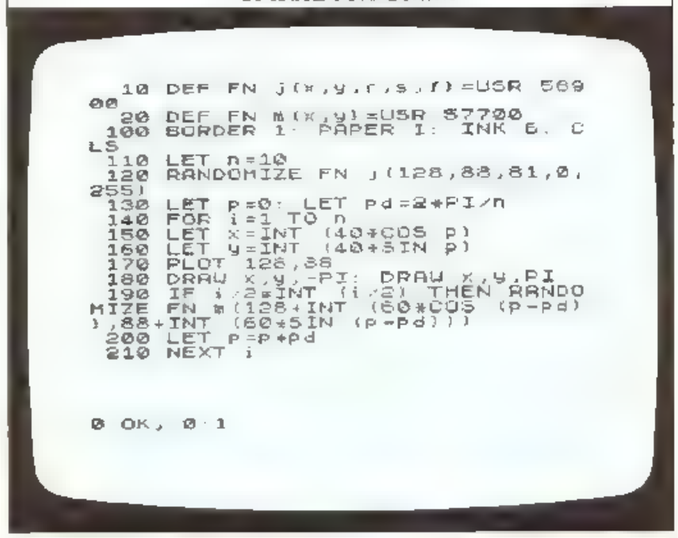
The spiral program is an effective use of the fill routine to colour in alternate portions of the circle. The two displays were achieved by varying n, which determines the number of spirals to be drawn.

SPIRAL PROGRAM

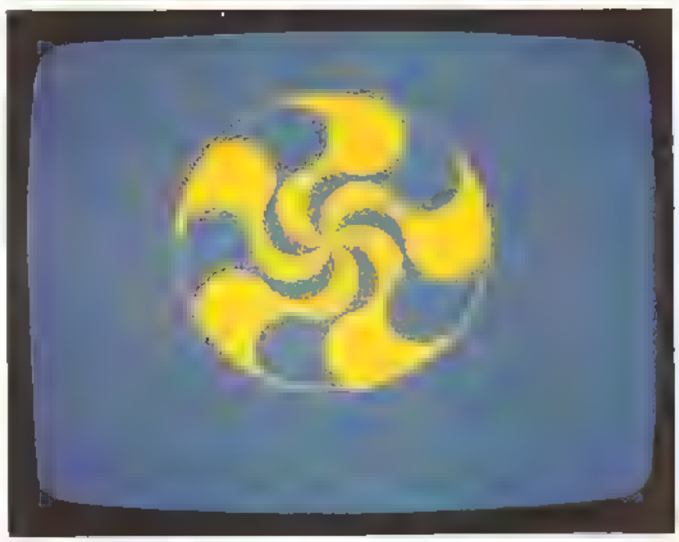
00:20 seconds

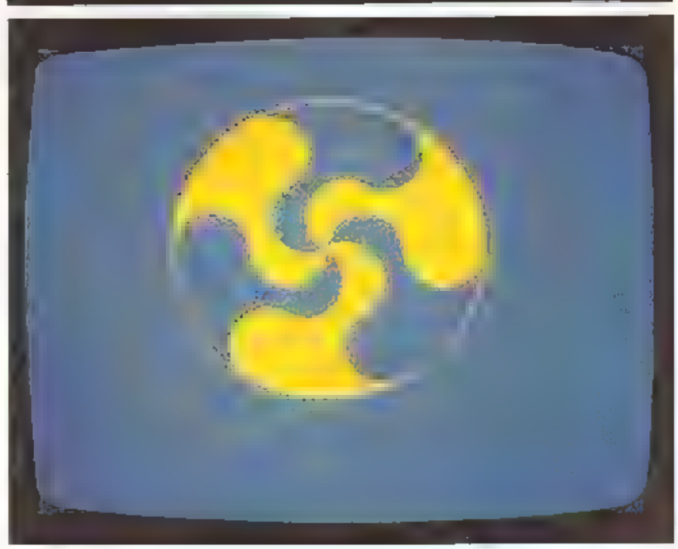
How the program works
Only the arc and fill routines are used in this program. A circle is drawn and then two BASIC semicircles are drawn to join the centre point to the circumference. After two of these curves have been drawn, the space between is filled and the sequence repeated. The number of spirals is set by variable n.
Lines 10-20 define the routines.
Line 120 draws a complete circle (centre 128,88).
Line 180 draws two curves in BASIC, using PI to specify semicircles.
Line 190 fills the area between two curves on alternate passes of the loop.

SPIRAL PROGRAM



```
  10 DEF FN j(x,y,r,s,f)=USR 589
00
  20 DEF FN m(x,y)=USR 57700
 100 BORDER 1: PAPER 1: INK 6: C
LS
 110 LET n=10
 120 RANDOMIZE FN j(128,88,81,0,
255)
 130 LET p=0: LET pd=2*PI/n
 140 FOR i=1 TO n
 150 LET x=INT (40*COS p)
 160 LET y=INT (40*SIN p)
 170 PLOT 128,88
 180 DRAW x,y,-PI: DRAW x,y,PI
 190 IF i/2=INT (i/2) THEN RANDO
MIZE FN m(128+INT (60*COS (p-pd)
),88+INT (60*SIN (p-pd)))
 200 LET p=p+pd
 210 NEXT i

0 OK, 0:1
```





OVERPRINTING AND ERASING

The OVER command in BASIC is one of four "logical operators" on the Spectrum; its more formal title is Exclusive/Or, or XOR for short. XOR forms the basis of the machine-code routine, FNn, on this page. You will recognize at once the other logical operators, since they occur in Spectrum BASIC with the same titles: AND, OR and NOT. Logical operators give a result depending on the way particular bits are set. The table below shows how the four operators make decisions.

TABLE OF LOGICAL OPERATORS

AND			OR			NOT		XOR		
A	B	**A AND B**	A	B	**A OR B**	A	**NOT A**	A	B	**A XOR B**
0	0	**0**	0	0	**0**	0	**1**	0	0	**0**
0	1	**0**	0	1	**1**	1	**0**	0	1	**1**
1	0	**0**	1	0	**1**			1	0	**1**
1	1	**1**	1	1	**1**			1	1	**0**

Thus, the XOR-line routine (FNn) looks at the screen before setting a pixel. If the pixel is currently set, the routine clears it; if the pixel is not set, however, the routine sets it.

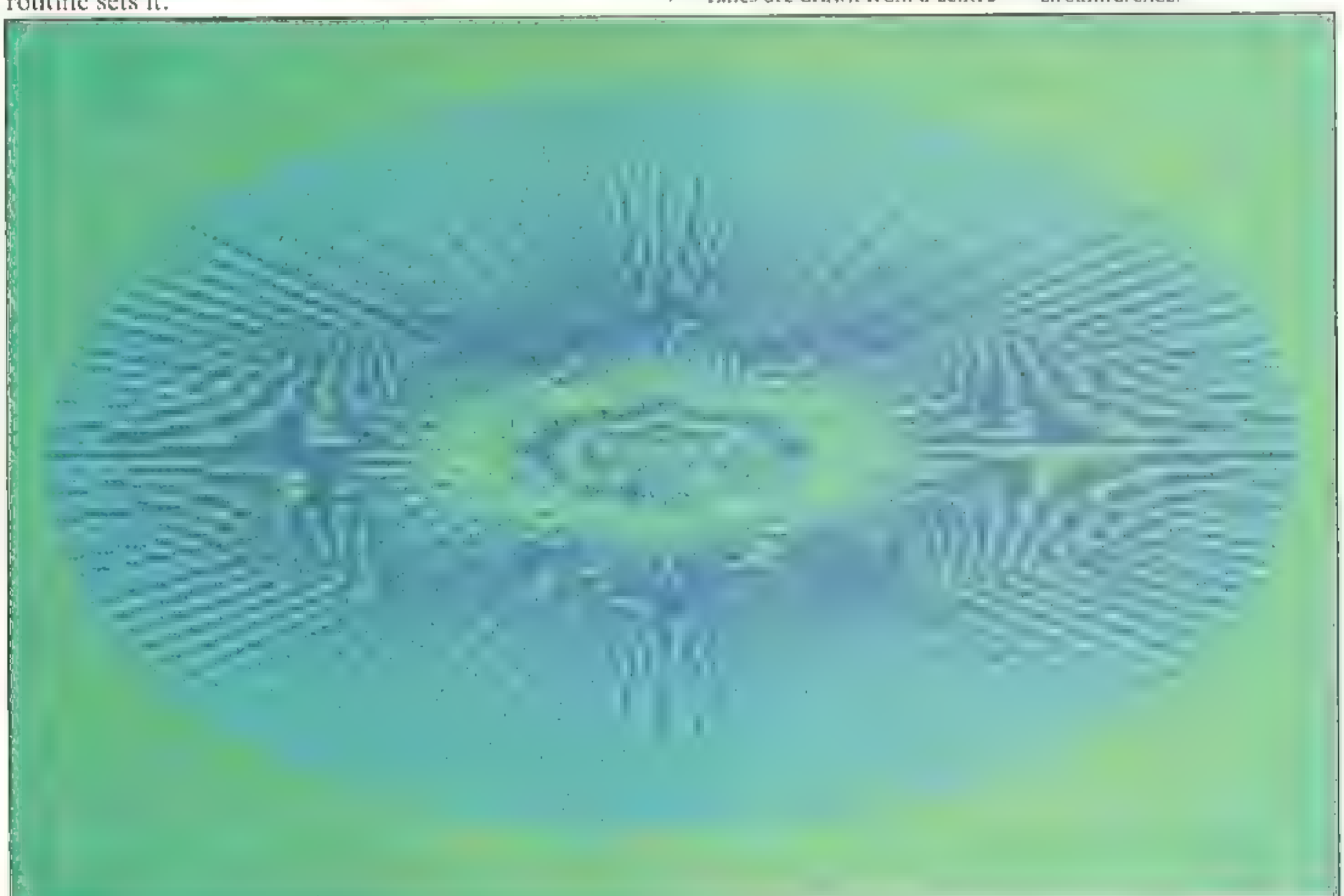
XOR ELLIPSE PROGRAM

```
  10 DEF FN n(x,y,p,q)=USR 57600
 100 BORDER ■: PAPER 5: INK 1: C
LS
 110 LET s=1: LET a=0: LET ad=s*
PI/128
 120 LET x1=127: LET y1=88
 130 FOR i=0 TO 255 STEP s
 140 LET x=x1+INT (110*SIN a)
 150 LET y=y1+INT (70*COS a)
 160 RANDOMIZE FN n(x,y,x1,y1)
 170 LET a=a+ad
 180 NEXT i
 190 PAUSE 0
 200 GO TO 110

0 OK, 0:1
```

XOR ELLIPSE PROGRAM

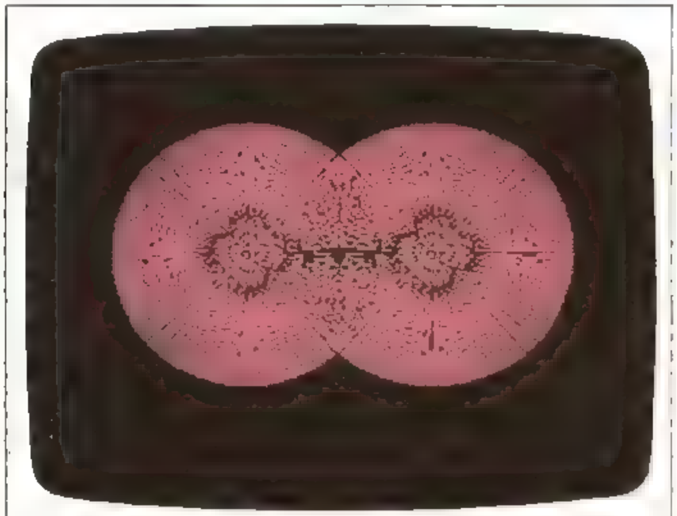
01:10 **minutes**

How the program works
Lines are drawn from a centre to points on an ellipse.
Line 130 specifies how many lines are to be drawn.
Lines 140-150 calculate the co-ordinates of a point on the circumference.



INTERFERENCE CIRCLES PROGRAM

```
  10 DEF FN n(x,y,p,q)=USR 57600
 100 BORDER 0: PAPER 0: INK 3: C
LS
 110 LET s=1: LET a=0: LET ad=s*
PI/128
 120 LET x1=80: LET y1=88
 130 FOR c=1 TO 2
 140 FOR i=0 TO 255 STEP s
 150 LET x=x1+INT (70*SIN a)
 160 LET y=y1+INT (70*COS a)
 170 RANDOMIZE FN n(x,y,x1,y1)
 180 LET a=a+ad
 190 NEXT i
 200 LET x1=x1+96
 210 NEXT c
 220 PAUSE 100
 230 GO TO 110

0 OK, 0:1
```



The interference circles program shows how, by using XOR lines, two overlapping circles can produce an interesting pattern instead of an area of solid colour.

OVERPRINTING PROGRAM

```
  10 DEF FN n(x,y,p,q)=USR 57600
 100 BORDER 1: PAPER 6: INK 2
 110 CLS
 120 FOR i=5 TO 15
 130 PRINT AT i,3;"123456      654
321"
 140 NEXT i
 150 FOR j=18 TO 238 STEP 3
 160 RANDOMIZE FN n(128,90,j,10)
 170 RANDOMIZE FN n(128,90,j,168
)
 180 NEXT j
 190 FOR j=10 TO 168 STEP 10
 200 RANDOMIZE FN n(129,90,18,j)
 210 RANDOMIZE FN n(128,90,238,j
)
 220 NEXT j
 230 PAUSE 0: GO TO 150

0 OK, 0:1
```

FNn

XOR-LINE ROUTINE

Start address 57600 **Length** 20 bytes
Other routines called Line-draw routine (FNg).
What it does Draws an Exclusive/OR line on the screen between two specified points.

Using the routine This routine works in the same way as the line-draw routine, except that Exclusive/OR allows you to erase what has been drawn. Using the routine you can draw lines over an image and then remove them again, without affecting the original image. As for the line-draw routine, the routine incorporates some error-trapping.

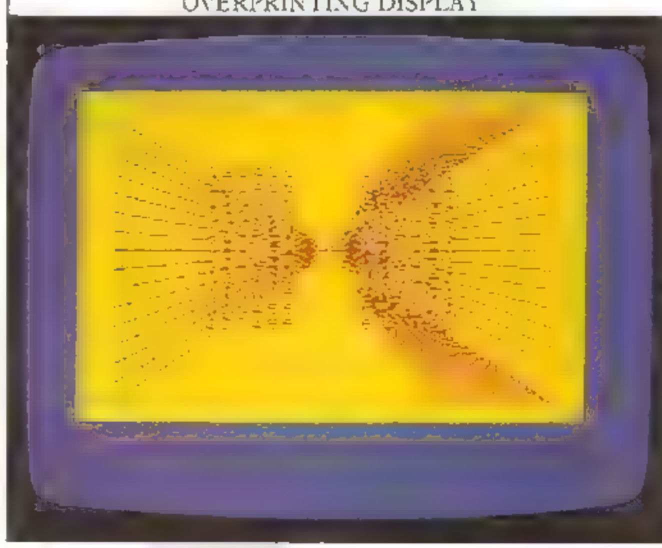
ROUTINE PARAMETERS

DEF FN n(x,y,p,q)	
x,y	specify the start pixel co-ordinates of the XOR-line ($x<256, y<176$)
p,q	specify the end pixel co-ordinates of the XOR-line ($p<256, q<176$)

ROUTINE LISTING

```
8050 LET b=57600: LET l=15: LET
z=0: RESTORE 8060
8051 FOR i=0 TO l-1: READ a
8052 POKE (b+i),a: LET z=z+a
8053 NEXT i
8054 LET z=INT (((z/l)-INT (z/l)
)*l)
8055 READ a: IF a<>z THEN PRINT
"??": STOP

8060 DATA 62,168,50,223,237
8061 DATA 205,28,237,62,176
8062 DATA 50,223,237,201,0
8063 DATA 13,0,0,0,0
```

OVERPRINTING DISPLAY



Finally, the overprinting program gives an example of the XOR-line routine being used to draw over some text and cover it (lines 160-170), and then "undraw" the lines by calling the routine again in lines 200 and 210, leaving the text intact.

COMBINING ROUTINES

The programs on this page give some further examples of combining the routines used earlier in this book. You will see from the programs used here that, in a program of any length, it is a good idea to separate the machine-code routines clearly at the beginning of the program, as has been done here.

Although the programs look complicated, they both consist mainly of machine-code calls. The repeated circles program is a symmetrical pattern; the small circles on the circumference of the large ones are drawn in lines 230-380. Variables x,y, which are points on the circumference of a large circle of radius rz, are used to determine the centre of the small circles. The actual centre points of the small circles are obtained by adding or subtracting an offset (x0,y0) from x and y in lines 270-340. Variables xm,ym are used to calculate the co-ordinates for the fill routine.

The kite program is even simpler; the only complicated part is the drawing of the tail (drawn by a subroutine in lines 500-600). The number of bows in the tail can be modified by changing the variable s in line 110.

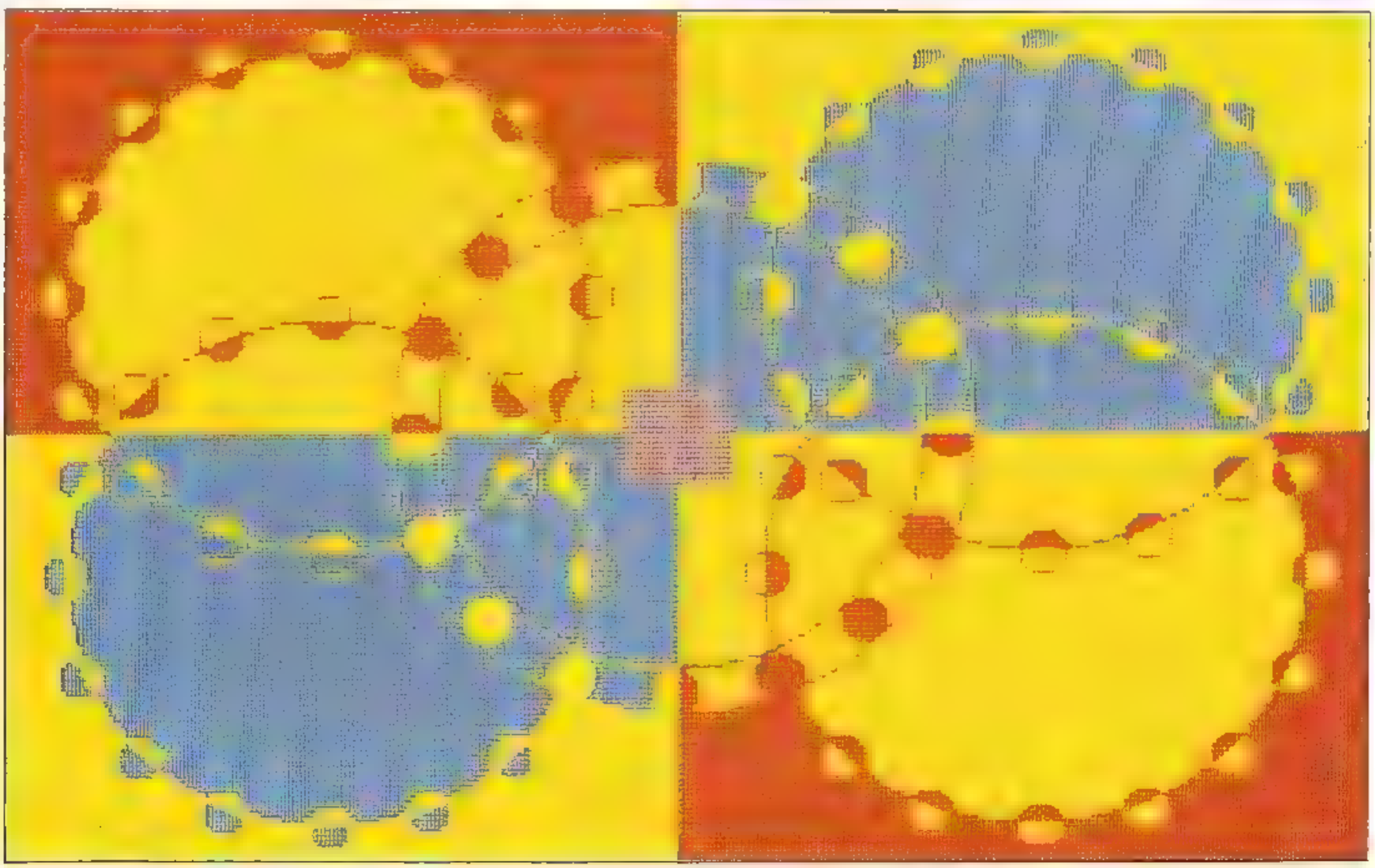
REPEATED CIRCLES PROGRAM

00:18 **seconds**

How the program works
This program displays circles with smaller circles on their circumference. Each of the small circles is then half-filled.
Line 100 defines ad, the step size.
Line 120 defines x0 and y0, the offset from the centre for the four large circles, and rx and rz, the radius of the small and large circles.
Lines 140-200 draw the large circles and the centre box.
Lines 250-380 draw the small circles.
Lines 400-440 set the colours of the four quarters of the screen.

REPEATED CIRCLES PROGRAM

```
  10 DEF FN b(x,y,h,v,c,b,f)=USR
 62800
  20 DEF FN c(x,y,h,v,c,b,f)=USR
 62600
  30 DEF FN h(x,y,h,v)=USR 60400
  40 DEF FN j(x,y,r,s,f)=USR 589
00
  50 DEF FN m(x,y)=USR 57700
 100 LET ad=16+PI/128: LET a=0
 110 LET x1=127: LET y1=87
 120 LET xo=66: LET yo=27: LET r
z=50: LET rx=5
 130 BORDER 0: PAPER 1: INK 6: C
LS
 140 RANDOMIZE FN j(x1-xo,y1-yo,
rz,0,255)
 150 RANDOMIZE FN j(x1+xo,y1-yo,
rz,0,255)
 160 RANDOMIZE FN j(x1+xo,y1+yo,
rz,0,255)
 170 RANDOMIZE FN j(x1-xo,y1+yo,
rz,0,255)

scroll?
```



REPEATED CIRCLES PROGRAM CONTD.

```
 180 RANDOMIZE FN j(x1,y1,rz+rx,
0,255)
 190 RANDOMIZE FN j(x1,y1,rz-rx+
1,0,255)
 200 RANDOMIZE FN h(117,76,20,16
)
 210 RANDOMIZE FN m(128,88)
 220 FOR i=0 TO 255 STEP 16
 230 LET x=x1+INT (rz*SIN a)
 240 LET y=y1+INT (rz*COS a)
 250 LET xm=x1+INT ((rz-3)*SIN a
)
 260 LET ym=y1+INT ((rz-3)*COS a
)
 270 RANDOMIZE FN j(x-xo,y-yo,rx
,0,255)
 280 RANDOMIZE FN m(xm-xo,ym-yo)
 290 RANDOMIZE FN j(x+xo+1,y-yo,
rx,0,255)
 300 RANDOMIZE FN m(xm+xo+1,ym-y
o)
 310 RANDOMIZE FN j(x+xo+1,y+yo,

scroll?
```

```
rx,0,255)
 320 RANDOMIZE FN m(xm+xo+1,ym+y
o)
 330 RANDOMIZE FN j(x-xo,y+yo,rx
,0,255)
 340 RANDOMIZE FN m(xm-xo,ym+yo)
 350 RANDOMIZE FN j(x,y,rx,0,255
)
 360 RANDOMIZE FN m(xm,ym)
 370 LET a=a+ad
 380 NEXT i
 390 RANDOMIZE FN m(1,1)
 400 RANDOMIZE FN b(0,0,16,11,2,
0,0)
 410 RANDOMIZE FN b(16,11,16,11,
2,0,0)
 420 RANDOMIZE FN c(0,0,16,11,6,
0,0)
 43█ RANDOMIZE FN c(16,11,16,11,
6,0,0)
 440 RANDOMIZE FN b(14,9,4,4,3,0
,0)

0 OK, 0:1
```

KITE PROGRAM

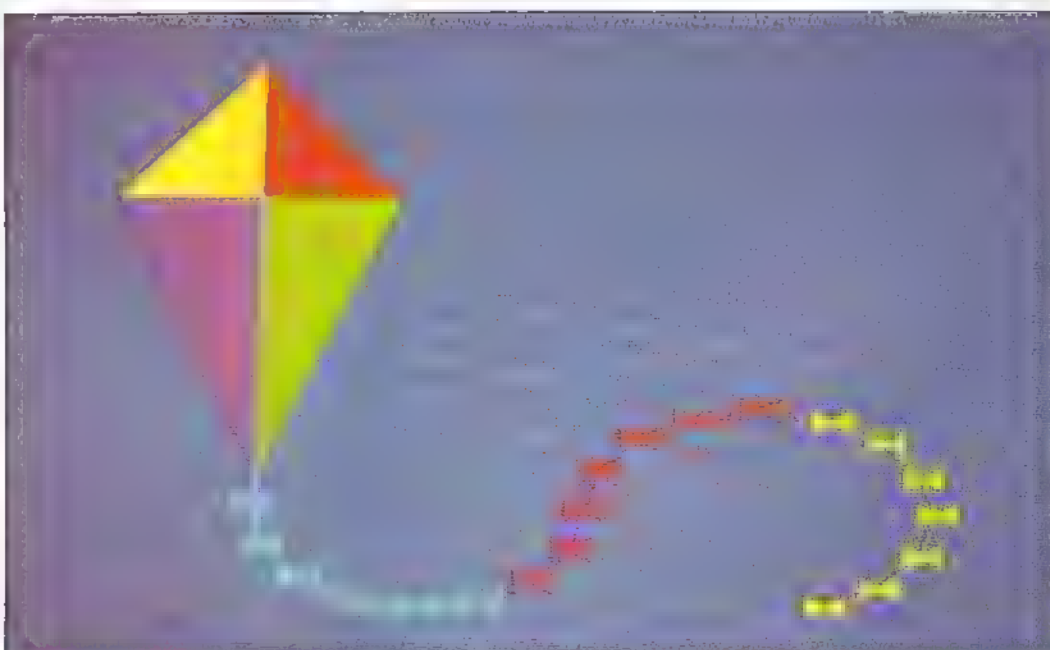
```
  10 DEF FN b(x,y,h,v,c,b,f)=USR
 62800
  20 DEF FN g(x,y,p,q)=USR 60700
  30 DEF FN i(x,y,p,q,r,s)=USR 6
0300
  40 DEF FN m(x,y)=USR 57700
 100 BORDER 1: PAPER 1: INK 2: C
LS
 110 LET s=16: LET ad=s*PI/128:
LET a=ad*64/s: LET lt=128: LET x
1=47: LET y1=127
 120 RANDOMIZE FN i(x1-40,y1,x1+
40,y1,x1,y1+40)
 130 RANDOMIZE FN i(x1-40,y1,x1+
40,y1,x1,y1-80)
 140 RANDOMIZE FN m(x1,y1)
 150 LET x2=143: LET y2=40: LET
x1=x1+47: LET y1=y1-90
 160 GO SUB 500
 170 RANDOMIZE FN g(47,50,47,40)
 180 LET x1=x1+96: LET lt=172: L
ET x2=143: LET y2=40: GO SUB 500

scroll?
```

```
 190 RANDOMIZE FN b(1,1,5,5,6,0,
0): RANDOMIZE FN b(6,6,5,10,4,0,
0): RANDOMIZE FN b(1,6,5,10,3,0,
0)
 200 RANDOMIZE FN b(25,8,6,14,5,
0,0)
 210 RANDOMIZE FN b(5,16,10,6,5,
0,0)
 220 PAUSE 0: STOP
 500 FOR i=0 TO lt STEP s
 510 LET x=x1+INT (48*SIN a)
 520 LET y=y1+INT (30*COS a)
 530 RANDOMIZE FN i(x,y,x-5,y+3,
x-5,y-3)
 540 RANDOMIZE FN m(x-2,y)
 550 RANDOMIZE FN i(x,y,x+5,y+3,
x+5,y-3)
 560 RANDOMIZE FN m(x+2,y)
 570 RANDOMIZE FN g(x,y,x2,y2)
 580 LET x2=x: LET y2=y
 590 LET a=a+ad
 600 NEXT i: RETURN

0 OK, 0:1
```

KITE PROGRAM

00:07 **seconds**

How the program works
The program draws a kite using coloured triangles, and then adds a tail with bows.
Lines 120-130 draw the kite using triangles.
Lines 150 and 180 set values for the subroutine variables.
Line 190 sets colours for the tail.
Line 500 is the start of the tail subroutine.
Lines 510 and 520 calculate the point x,y, at which an ellipse is drawn.
Lines 530-560 draw and fill in each bow.
Line 570 draws a line between each bow.
Lines 580 and 590 set values for the next bow to be drawn.



GRAPHICS EDITOR 1

Perhaps the most effective way of using machine-code routines like those in this book is in a single program which enables you to use the routines together. Although by this stage in the book you have enough routines available to create the kind of sophisticated displays seen in much commercial software, you do not have what the professionals use: a complete graphics editor. This is the purpose of the following program.

The graphics editor program

Each stage of the editor program incorporates routines from this book. The final program includes a facility for SAVEing and LOADing individual screens. The displays accompanying the program on this page and on the following few pages will give you some idea of the sort of pictures you can produce using the completed program.

How the program is built up

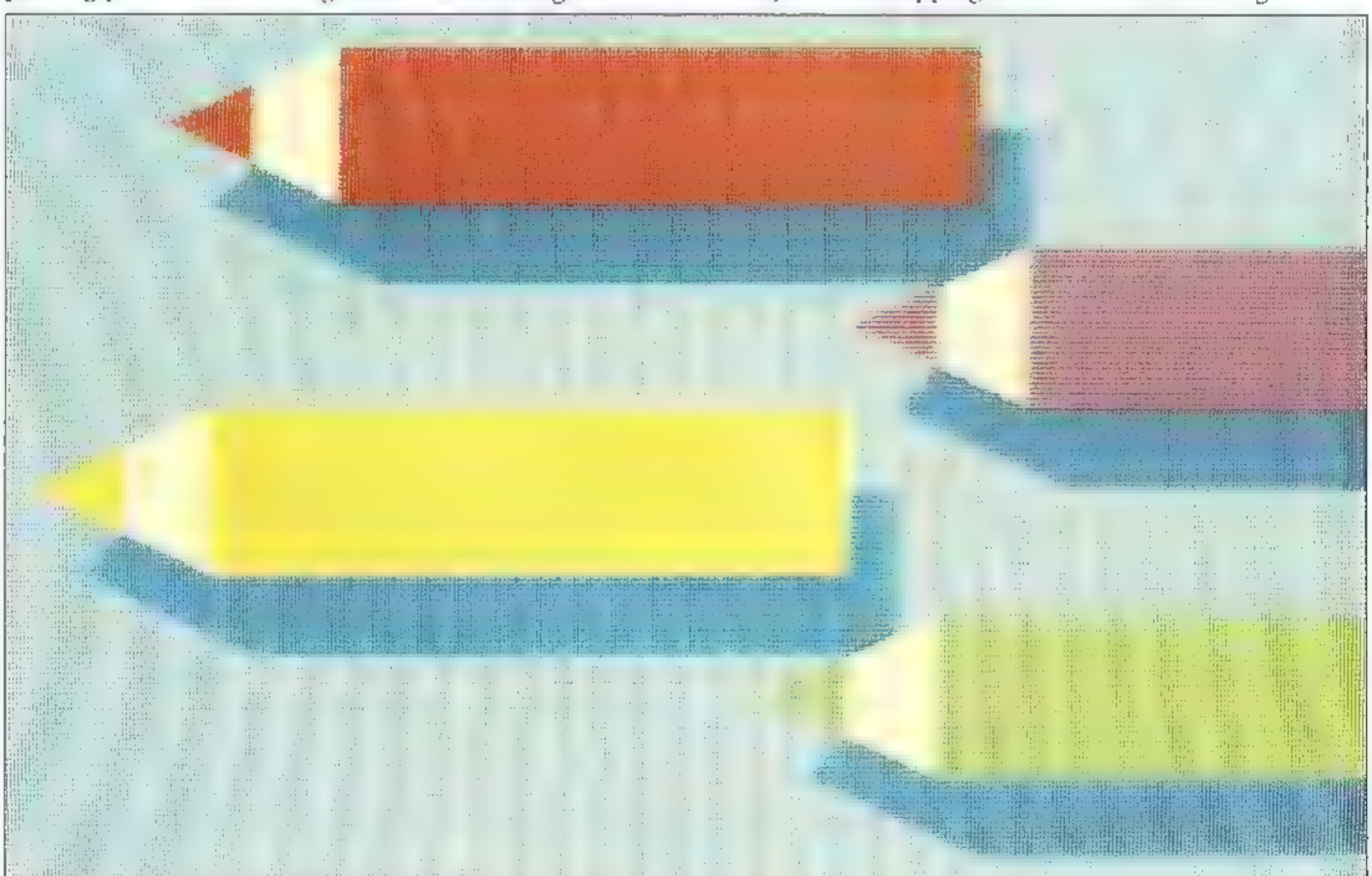
The graphics editor is shown in five stages, with each stage complete in itself. By keying in the lines on this page, you will have enough of the program to be able to move two cursors on the screen. These are used for plotting points and drawing lines in future stages.

GRAPHICS EDITOR STAGE 1

```
  10 DEF FN n(x,y,p,q)=USR 57600
  50 DEF FN h(x,y,h,v)=USR 60400
 100 PAPER 0: BORDER 0: INK 7: C
LS
 110 RANDOMIZE FN h(0,0,255,175)
 120 GO TO 1000
 200 LET cl=20: LET cr=20
 210 IF cx<20 THEN LET cl=cx: GO
 TO 230
 220 IF cx>235 THEN LET cr=255-c
x
 230 RANDOMIZE FN n(cx-cl,cy,cx+
cr,cy)
 240 LET cl=20: LET cr=20
 250 IF cy<20 THEN LET cl=cy: GO
 TO 270
 260 IF cy>155 THEN LET cr=175-c
y
 270 RANDOMIZE FN n(cx,cy-cl,cx,
cy+cr)
 280 RETURN
 300 LET me=7: LET mw=7: LET mu=

scroll?
```

How stage one works

Only two machine-code routines are used in stage one. The box-draw routine, FNh, draws a line round the edge of the drawing area, to prevent the fill routine (added later) from "wrapping around" the screen edge.



GRAPHICS EDITOR STAGE 1 CONTD.

```
7: LET md=7
 310 IF mx<7 THEN LET mw=0: LET
mu=mw: GO TO 330
 320 IF mx>248 THEN LET me=0: LE
T md=me
 330 IF my<mw OR my<md THEN LET
mw=0: LET md=mw: GO TO 350
 340 IF 175-my<mu OR 175-my<me T
HEN LET mu=0: LET me=mu
 350 RANDOMIZE FN n(mx-mw,my-mw,
mx+me,my+me)
 360 RANDOMIZE FN n(mx+md,my-md,
mx-mu,my+mu)
 370 RETURN
 900 LET mx=cx: LET my=cy
 910 GO SUB cur: GO SUB mar
 920 GO TO 1100
1000 LET kol=0: LET ink=0: LET p
ap=7: LET d=0: LET cur=200: LET
mar=300: LET g=0: LET b=0: LET f
l=0: LET f=255
1010 LET cx=160: LET cy=100: LET

scroll?
```

```
 mx=20: LET my=110
1020 GO SUB cur: GO SUB mar
1100 LET a$=INKEY$
1110 IF a$="" THEN LET d=0: GO T
O 1100
1120 LET ke=CODE a$: LET d=d+1
1130 IF ke<>9 THEN GO TO 1170
1140 GO SUB cur: LET cx=cx+d
1150 IF cx>255 THEN LET cx=255:
LET d=0
1160 GO SUB cur: GO TO 1100
1170 IF ke<>8 THEN GO TO 1210
1180 GO SUB cur: LET cx=cx-d
1190 IF cx<0 THEN LET cx=0: LET
d=0
1200 GO SUB cur: GO TO 1100
1210 IF ke<>10 THEN GO TO 1250
1220 GO SUB cur: LET cy=cy-d
1230 IF cy<0 THEN LET cy=0: LET
d=0
1240 GO SUB cur: GO TO 1100
1250 IF ke<>11 THEN GO TO 1290

scroll?
```

```
1260 GO SUB cur: LET cy=cy+d
1270 IF cy>175 THEN LET cy=175:
LET d=0
1280 GO SUB cur: GO TO 1100
1290 IF ke<>77 THEN GO TO 1320
1300 GO SUB mar: LET mx=cx: LET
my=cy
1310 GO SUB mar: GO TO 1100
1320 GO TO 1100

0 OK, 0:1
```

The other routine included here is the XOR line routine, FNn, used to draw the two cursors. Exclusive OR plotting is used because the cursors have to be able to move around the screen and remain visible, without disturbing whatever has already been drawn. Try moving one of the cursors in this program to a corner of the screen to see the XOR effect. The screen border remains unchanged when the cursor is moved away.

Line 1000 is the beginning of the main routine. It gives initial values to all the variables used in the program. These, for example, store values for INK,

CURSORS DISPLAY

PAPER, FLASH and BRIGHT. After setting initial co-ordinates for the two cursors (points cx,cy and mx,my), the program moves to the subroutines. These are stored early in the program to increase the running speed.

The cursor subroutine is at lines 200-280, and the cursor is positioned at point cx,cy. The second cursor is placed on the screen using the subroutine at lines 300-370 (points mx,my). These cursor subroutines (called cur, mar) are used to delete the cursors before any routine is called, and again to put the cursors back on the screen afterwards.

GRAPHICS EDITOR PARAMETERS

A	attribute edit	**L**	line
I	ink		
P	paper	**Q**	window paper
O	bright/flash		
ENTER	to quit attribute edit	**S**	save screen
B	box	**T**	triangle (press T again for second corner of triangle)
C	circle		
S	start		
F	finish	**W**	partial screen clear
D	dot	**X**	text
		ENTER	to quit text
E	window ink		
F	fill		
G	grid		
J	load screen		

All these instructions require you to press CAPS SHIFT followed by the letter shown, in upper case.

GRAPHICS EDITOR 2

The second stage of the graphics editor adds routines for points, lines and boxes, as well as adding the ink, paper and partial screen clear routines.

Colour is set by the subroutines in lines 400-850. These allow you to select colour, BRIGHT and FLASH values. Points are drawn using the point-plot routine, in lines 1320 to 1350. Some of the details in these lines reappear throughout the program. Line 1320, for example, checks to see if key D has been pressed (ASCII code number 68). If it has, the two cursor subroutines are called, and the values of cx and cy are used as the co-ordinates of the point to be plotted. Lines 1360 to 1490 work in a similar way for the line draw, fill and box routines. Line 1500 is a "dummy" line, where other routines will be inserted.

The grid subroutine

Lines 1730 to 1790 set up a grid on the screen, by

GRAPHICS EDITOR STAGE 2

```
  20 DEF FN f(x,y)=USR 61500
  30 DEF FN g(x,y,p,q)=USR 50700
  40 DEF FN m(x,y)=USR 57700
  80 DEF FN b(x,y,h,v,c,b,f)=USR
62800
  90 DEF FN c(x,y,h,v,c,b,f)=USR
62600
  95 DEF FN a(x,y,h,v)=USR 63000
 120 GO SUB 6000: GO TO 1000
 400 PRINT #0;"0-7?"
 410 PAUSE 0: LET a$=INKEY$: IF
a$<"0" OR a$>"7" THEN GO TO 410
 420 INPUT "": LET c=VAL a$
 430 PRINT #0,"Bright?"
 440 PAUSE 0: LET a$=INKEY$: IF
a$<"0" OR a$>"7" THEN GO TO 440
 450 INPUT "": LET b=VAL a$
 460 PRINT #0,"Flash?"
 470 PAUSE 0: LET a$=INKEY$: IF
a$<"0" OR a$>"7" THEN GO TO 470
 480 INPUT "": LET fl=VAL a$
 490 RETURN

scroll?
```

```
 600 LET xc=INT (cx/8): LET yc=2
1-INT (cy/8)
 610 LET xm=INT (mx/8): LET ym=2
1-INT (my/8)
 620 LET x=xc: IF xc>xm THEN LET
 x=xm
 630 LET y=yc: IF yc>ym THEN LET
 y=ym
 640 LET h=ABS (xc-xm)+1: LET v=
ABS (yc-ym)+1
 650 RETURN
1320 IF ke<>68 THEN GO TO 1360
1330 GO SUB cur: GO SUB mar
1340 RANDOMIZE FN f(cx,cy)
1350 GO TO 900
1360 IF ke<>76 THEN GO TO 1400
1370 GO SUB cur: GO SUB mar
1380 RANDOMIZE FN g(mx,my,cx,cy)
1390 GO TO 900
1400 IF ke<>70 THEN GO TO 1440
1410 GO SUB cur: GO SUB mar
1420 RANDOMIZE FN m(cx,cy): LET

scroll?
```

GRAPHICS EDITOR STAGE 2 CONTD.

```
mx=cx: LET my=cy
1430 GO TO 910
1440 IF ke<>66 THEN GO TO 1500
1450 GO SUB cur: GO SUB mar
1460 LET x=cx: IF cx>mx THEN LET
 x=mx
1470 LET y=cy: IF cy>my THEN LET
 y=my
1480 RANDOMIZE FN h(x,y,ABS (cx-
mx),ABS (cy-my))
1490 GO TO 900
1500 GO TO 1730
1730 IF ke<>71 THEN GO TO 1800
1740 IF g=1 THEN GO TO 1780
1750 LET g=1
1760 RANDOMIZE USR 55500
1770 GO TO 1100
1780 RANDOMIZE USR 55531
1790 LET g=0: GO TO 1100
2140 IF ke<>81 THEN GO TO 2190
2150 GO SUB 600
2160 GO SUB 400: GO SUB cur: GO

scroll?
```

printing character squares in normal and BRIGHT alternately. When key G is pressed, a grid drawn by a machine-code routine appears. The routine is POKEd into memory by the subroutine at lines 6000 (called in line 120). The grid is used to show character borders on the screen while you are drawing a display.

GRAPHICS EDITOR STAGE 2 CONTD.

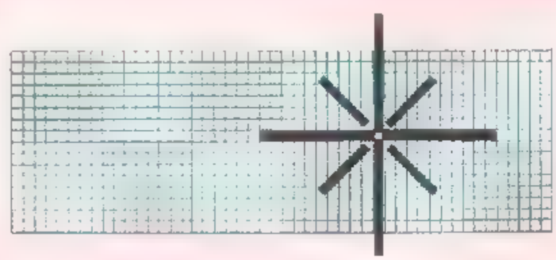
```
SUB mar
2170 RANDOMIZE FN c(x,y,h,v,c,b,
f())
2180 GO TO 910
2190 IF ke<>87 THEN GO TO 2240
2200 GO SUB 600
2210 GO SUB cur: GO SUB mar
2220 RANDOMIZE FN a(x,y,h,v)
2230 GO TO 910
2240 IF ke<>69 THEN GO TO 1100
2250 GO SUB 600
2260 GO SUB 400: GO SUB cur: GO
SUB mar
2270 RANDOMIZE FN b(x,y.h,v,c,b,
f())
2280 GO TO 910
6000 RESTORE 6030: FOR n=55500 T
O 55542
6010 READ a: POKE n,a
6020 NEXT n
6030 DATA 33,0,88,1,192,2,17,55,
217,237,176

scroll?
```

GRAPHICS EDITOR STAGE 2 CONTD.

```
6040 DATA 33,247,216,17,0,88,1,6
4,0,237,176
6050 DATA 33,0,88,1,128,2,237,17
6,201
6060 DATA 33,55,217,17,0,88,1,19
2,2,237,176,201
6070 FOR n=55543 TO 55574 STEP 2
6080 POKE n,120: POKE n+1,56
6090 NEXT n
6100 FOR n=55575 TO 55606 STEP 2
6110 POKE n,56: POKE n+1,120
6120 NEXT n
6130 RETURN

0 OK, 0:1
```

USING THE GRAPHICS EDITOR GRID

The graphics editor grid (CAPS SHIFT and G) is used to display character borders by setting the BRIGHT attributes of alternate characters. The grid does not delete anything currently on the screen. The cursor shows the current pixel position, superimposed on the grid. In the diagram, the cursor is on the leftmost pixel of a character square.



GRAPHICS EDITOR FILL DISPLAY

GRAPHICS EDITOR 3

The third stage of the graphics editor adds routines for drawing circles and triangles. These routines give you many new possibilities for your displays, as you can see from those shown here.

Drawing circles

Line 1500 is the start of the circle routine. Lines 1510 to 1530 enable the user to enter start and finish parameters between 0 to 360 degrees, rather than the machine code's 0-255 parameter values. Lines 1570 and 1580 contain some BASIC error-trapping to prevent a circle being drawn too far off the screen and causing the Spectrum to crash. These lines could be incorporated into any BASIC programs which call the circle routines.

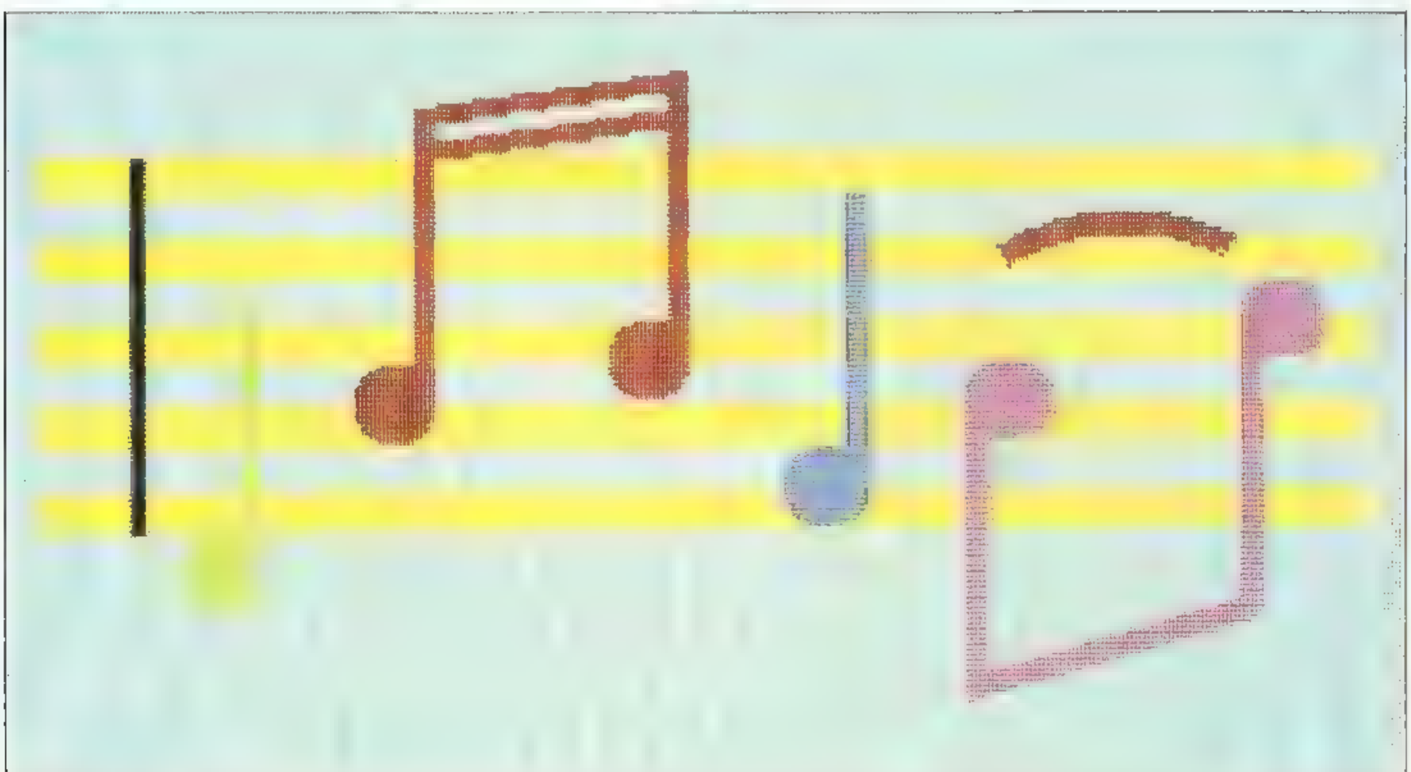
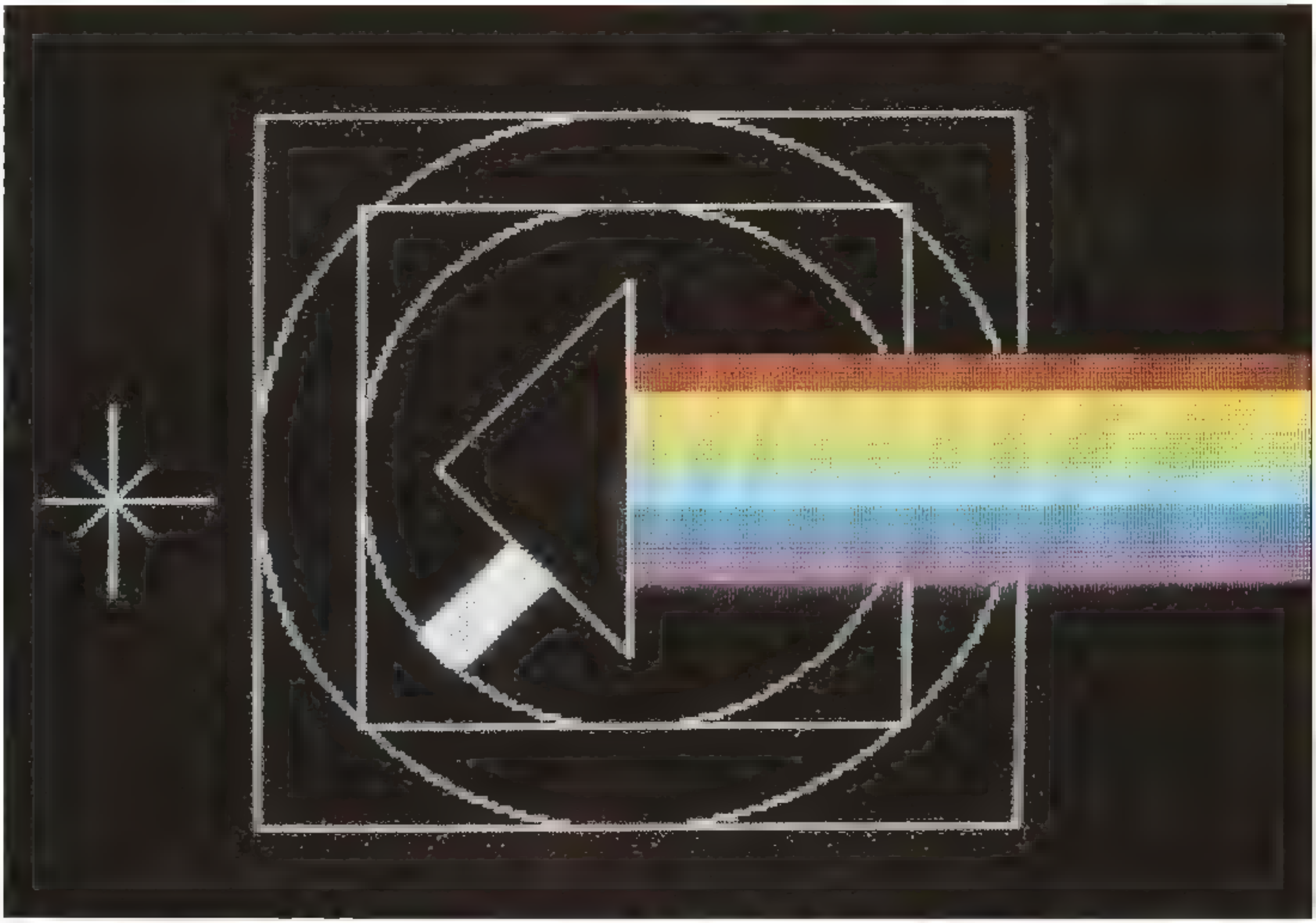
Adding triangles

Lines 1610 to 1720 are used to store corner co-ordinates for a triangle before calling the triangle routine, FNi. The parameters of the three corner points are held as variables cx,cy, mx,my and tx,ty.

GRAPHICS EDITOR STAGE 3

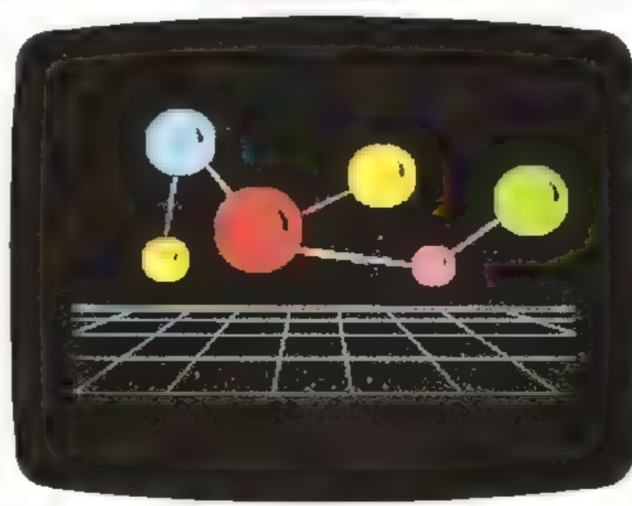
```
  60 DEF FN j(x,y,r,s,f)=USR 589
80
  70 DEF FN i(x,y,p,q,r,s)=USR 6
0300
1500 IF ke<>67 THEN GO TO 1610
1510 INPUT "s = ";s;"  f = ";f
1520 IF s<0 OR f<0 OR s>360 OR f
>360 OR s<>INT s OR f<>INT f THE
N GO TO 1510
1530 LET s=INT (255*(s/360)): LE
T f=INT (255*(f/360))
1540 GO SUB cur
1550 LET x=ABS (cx-mx): LET y=AB
S (cy-my)
1560 LET r=INT (SQR (x↑2+y↑2)+0.
5)
1570 IF r>255 THEN BEEP 2,3: INP
UT "": GO SUB cur: GO TO 1100
1580 IF r+mx>275 OR mx-r<-20 OR
r+my>195 OR my-r<-20 THEN LET r=
275: GO TO 1570
1590 RANDOMIZE FN j(mx,my,r,s,f)

scroll?
```

GRAPHICS EDITOR STAGE 3 CONTD.

```
1600 GO SUB cur: GO TO 1100
1610 IF ke<>84 THEN GO TO 1730
1620 LET p=mx: LET q=my: LET mx=
cx: LET my=cy: GO SUB mar
1630 LET a$=INKEY$: IF a$="" THE
N GO TO 1630
1640 LET a=CODE a$
1650 GO SUB cur
1660 LET cx=cx+1*(a=9 AND cx<255
)-1*(a=8 AND cx>0)
1670 LET cy=cy+1*(a=11 AND cy<17
5)-1*(a=10 AND cy>0)
1680 GO SUB cur
1690 IF a<>84 THEN GO TO 1630
1700 GO SUB cur: LET tx=mx: LET
ty=my: GO SUB mar: LET mx=p: LET
 my=q: GO SUB mar
1710 RANDOMIZE FN i(cx,cy,mx,my,
tx,ty)
1720 GO TO 900

0 OK, 0:1
```



BILLIARD BALL DISPLAY

SPACE STATION DISPLAY



Lines 1660 and 1670 take advantage of Spectrum BASIC's facility for writing conditional statements in an abbreviated form. Line 1660 could be rewritten as:

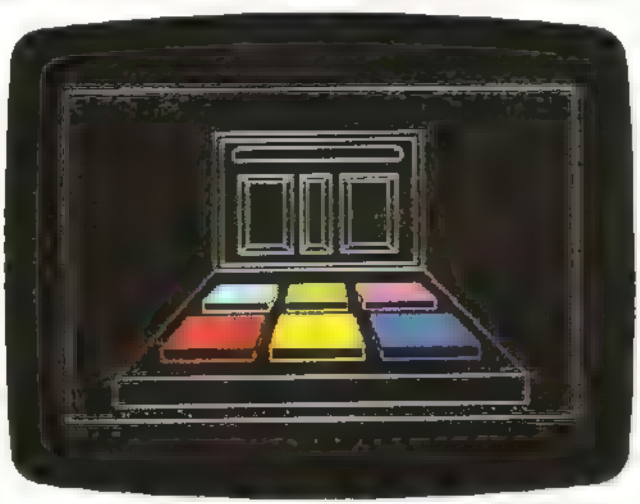
```
1660 IF a=9 AND cx<255 THEN LET cx=cx+1
1665 IF a=8 AND cx>0 THEN  LET cx=cx-1
```

These lines are used to move the cursor in BASIC to the third corner of the triangle, by calculating new values for cx and cy as a key is pressed. You will notice from the movement of this cursor how much slower BASIC movement is than the usual cursor speed, which is carried out by machine code. This speed advantage alone would be sufficient justification for using machine-code routines rather than BASIC.

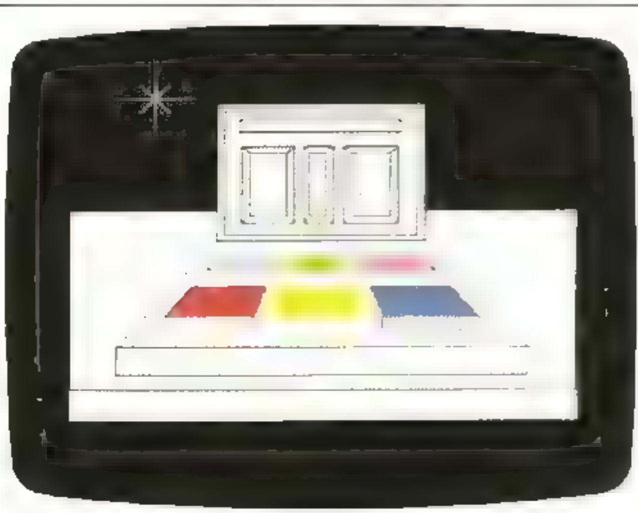
GRAPHICS EDITOR 4

The designs on this page show one method of producing a typical display. One general point is worth noting before beginning any large-scale graphics editor display. Each photograph of the cocktail display represents a point where the screen was SAVEd before going further to add more details. The reason for this is simple: even when you have a little experience with the editor, it is easy to ruin a display by adding an unintended line, or by filling a shape that is not totally

BLACK PAINTBOX DISPLAY



PAINTBOX DISPLAY



enclosed. Rather than risk losing an entire display, it is sensible to take a few seconds and SAVE what has been drawn before continuing.

The paintbox displays

The paintbox displays above show the difference in effect which can be obtained by drawing in white ink on a black background, rather than using black ink on a white background. This effect can be obtained by changing the initial graphics editor screen to black INK and white PAPER colours. A simple change like this can be very effective.

Building up a display

The cocktail display shown here is an example of how a graphics editor display can be developed in stages. Stage one of the display uses only lines, squares and triangles.

COCKTAIL DISPLAY 1

COCKTAIL DISPLAY 2

Even at this early stage, however, the design has been planned so that there will be no problem with character borders when colour is added. The position of character borders can of course be checked by using the grid (CAPS SHIFT and G). The grid does not delete anything which has been drawn, so it is a simple matter to flick between the grid and the normal screen as necessary at this stage to ensure that lines and points are

drawn in the correct position on either side of a character border.

The second stage uses circles and arcs to draw, for example, the cherry in the glass. Because of the relatively low resolution of the Spectrum screen display, a small circle such as that which forms the cherry may not be completely enclosed. Four single points were plotted on this circle to prevent the INK from "leaking" when the shape is filled. The umbrella in stage three was also drawn to take advantage of character borders when filled with colour. The colour change on the umbrella lies along a horizontal and vertical character border, although it appears from the display to be diagonal.
The picture was completed by filling areas and then adding colours. When drawing a complex display, it is always best to keep the filling and colouring operations until last. Remember also that colours should not be added while you are using the grid.

COCKTAIL DISPLAY 3

COCKTAIL DISPLAY 4

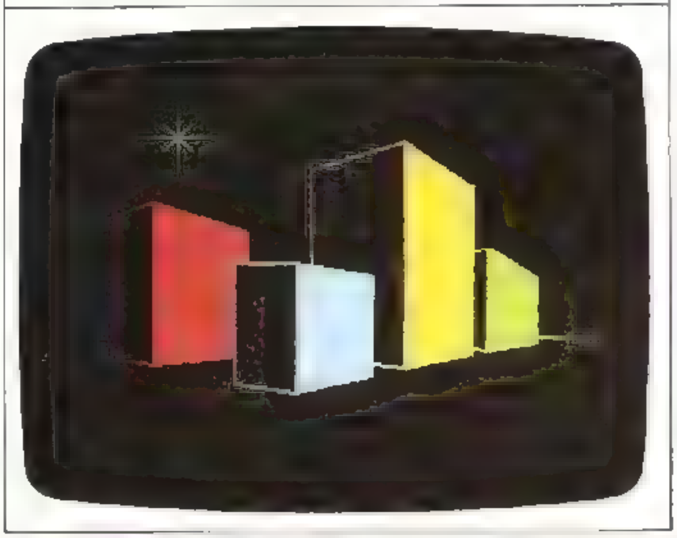
GRAPHICS EDITOR 5

Text is added by lines 1800-1890 of the program. Line 1810 takes the current pixel position (variables x,y) and converts these to character co-ordinates. This is because text is printed in character positions rather than using pixel co-ordinates. Line 1820 deletes the cursors, and line 1830 prints a flashing text prompt at the character position. Text is then entered in lines 1840-1870, which include BASIC controls for deleting mistakes in keying, and ending the text string when ENTER is pressed.

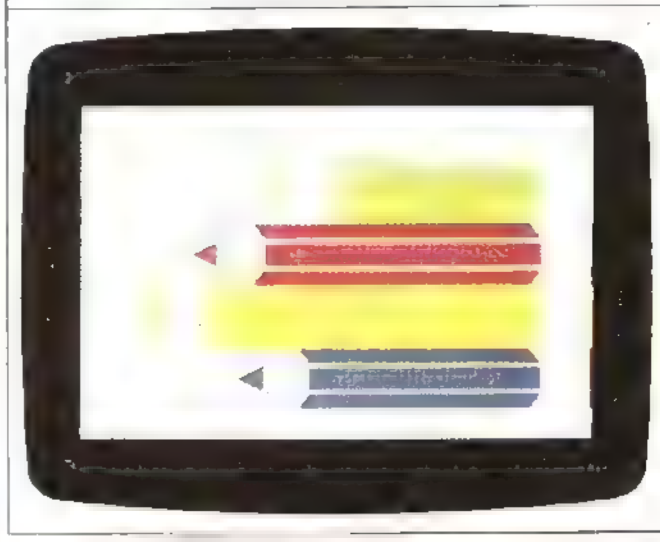
Attribute editing

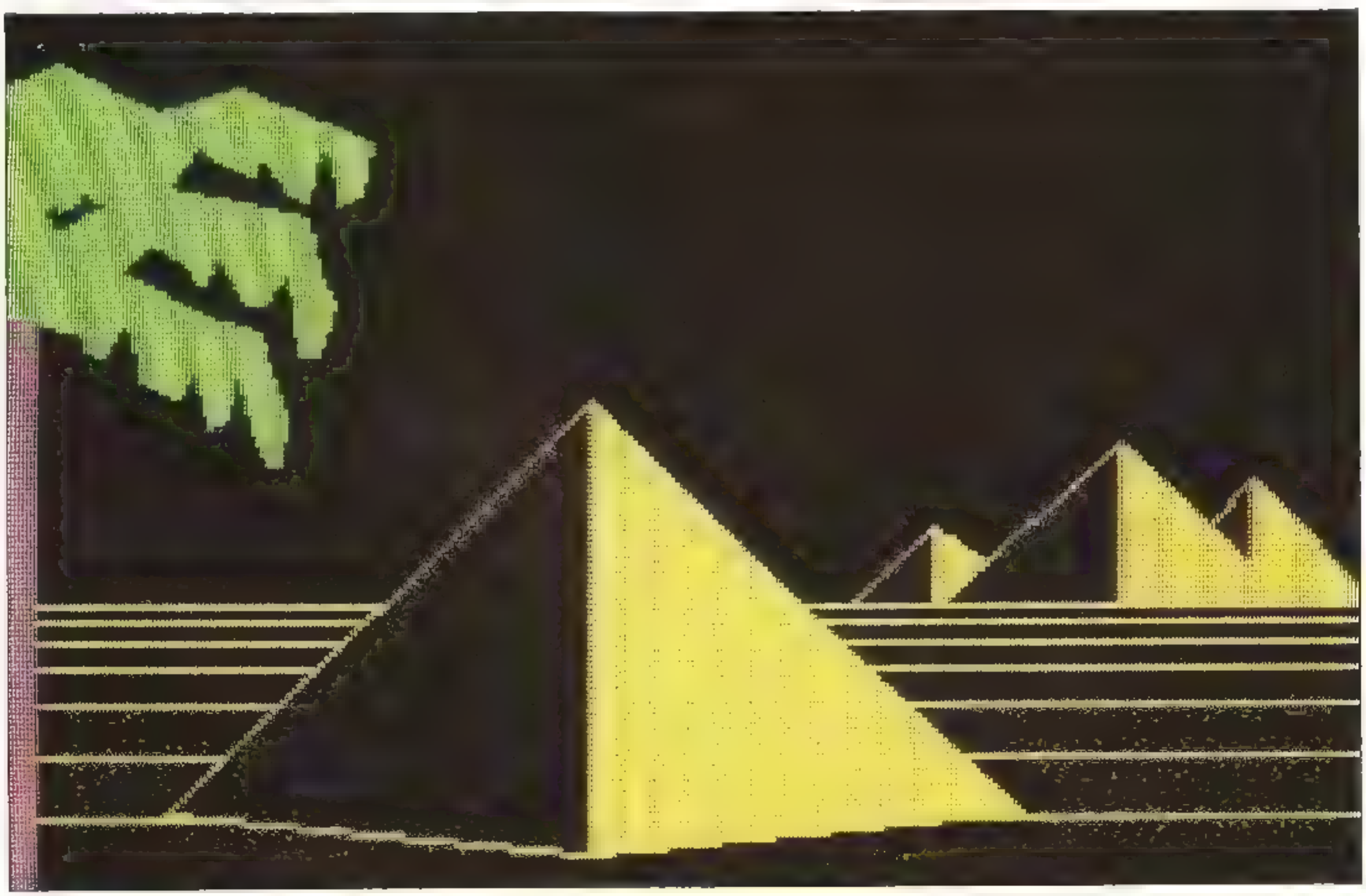
Lines 1900-2030 give you the option of setting INK,

BOXES DISPLAY



PENCILS DISPLAY





PAPER, BRIGHT and FLASH attributes of any character square on the screen. As with text, the current pixel position is converted to character co-ordinates (held as variables lin, col) for this routine. Lines 1970 and 1980 simply move the cursor (a flashing character square, printed in line 1960) onto the next line or column when the end of either is reached.

Saving and loading screens

Finally, lines 2040-2130 of the program enable you to SAVE and LOAD your displays, using the Spectrum SCREEN$ command. An advantage of this method is that you should be able to load onto the graphics editor the title display of many commercial games, since these programs often begin with a SCREEN$ display. This will enable you to make your own versions of these screens.

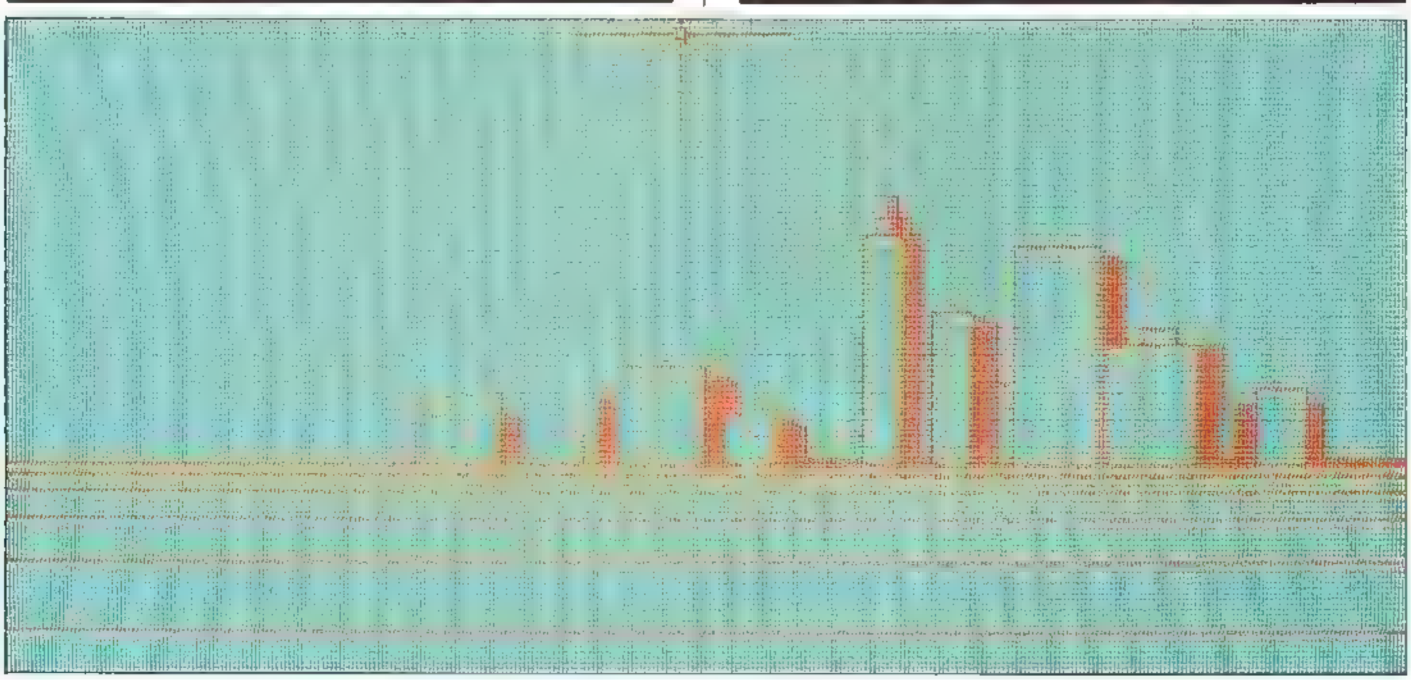
GRAPHICS EDITOR STAGE 5

```
 500 PRINT #0;"0-7 ?"
 510 LET a$=INKEY$  IF a$="" THE
N GO TO 510
 520 LET asc=CODE a$
 530 IF asc<47 OR asc>55 THEN GO
 TO 510
 540 INPUT ""
 550 RETURN
1800 IF ke<>88 THEN GO TO 1900
1810 LET x=INT (cx/8): LET y=21-
INT (cy/8)
1820 GO SUB cur: GO SUB mar
1830 PRINT AT y,x, OVER 1, FLASH
 1;">";CHR$ 8;
1840 PAUSE 0: LET a$=INKEY$: IF
a$="" OR a$<CHR$ 12 THEN GO TO 1
840
1850 IF a$=CHR$ 13 THEN GO TO 18
60
1860 IF a$=CHR$ 12 THEN PRINT CH
R$ 8+"  "+CHR$ 8+CHR$ 8;: LET a$
=""

scroll?
```

GRAPHICS EDITOR STAGE 5 CONTD.

```
1870 PRINT a$; OVER 1; FLASH 1;"
>";CHR$ 8;: GO TO 1840
1880 PRINT OVER 1;">": GO TO 910
1890 GO TO 910
1900 IF ke<>65 OR g=1 THEN GO TO
 2040
1910 GO SUB cur: GO SUB mar
1920 LET col=INT (cx/8): LET lin
=21-INT (cy/8)
1930 PRINT AT lin,col; INK ink;
PAPER pap; OVER 1; FLASH 1;" "
1940 LET a$=INKEY$: IF a$="" THE
N GO TO 1940
1950 LET asc=CODE a$
1960 PRINT AT lin,col; INK ink;
PAPER pap; BRIGHT b; FLASH fl; O
VER 1;" "
1970 LET col=col+1*(col<31 AND a
sc=9)-1*(col>0 AND asc=8)
1980 LET lin=lin+1*(lin<21 AND a
sc=10)-1*(lin>0 AND asc=11)
1990 IF asc=13 THEN GO TO 910

scroll?
```

```
2000 IF asc=73 THEN GO SUB 500:
LET ink=asc-48
2010 IF asc=80 THEN GO SUB 500:
LET pap=asc-48
2020 IF asc=79 THEN GO SUB 430
2030 GO TO 1930
2040 IF ke<>83 THEN GO TO 2090
2050 INPUT "SAVE ";p$
2060 GO SUB cur: GO SUB mar
2070 INPUT "": SAVE p$SCREEN$
2080 GO TO 910
2090 IF ke<>74 THEN GO TO 2140
2100 INPUT "LOAD ",p$
2110 GO SUB cur: GO SUB mar
2120 LOAD p$SCREEN$
2130 GO TO 910

0 OK, 0:1
```



MULTIPLE LINES

When using the line-draw routine (FNg), you must specify both the start and the end points for each line drawn. Where only a few lines are involved, this is not difficult, but if you are drawing a complicated shape with many lines joined together, you will find yourself continually specifying each point twice: once as the end of a line, and then again as the start of the next line. This can be avoided by using the multiple line-draw routine (FNo). This routine takes a series of co-ordinates which have been stored in memory, and joins each point in turn to the one before.

Having drawn your complex series of lines so quickly, you now need a way of wiping them off without damaging the rest of the display. For this reason an Exclusive/OR version of the routine is also included (FNp). This routine is the same as the multiple line-draw routine, but plots XOR lines. The XOR routine will enable you to repeatedly draw and undraw a whole series of lines on the screen in a few seconds.

Putting the points in memory

Before using the routines, you must specify the co-ordinates of the points to be linked, which are stored in a buffer. In operation, the routine takes a point from memory and joins it to the next point, and continues until it reaches a y co-ordinate of 255. Points can be POKEd into memory by using a loading routine such as the one below, which accepts pairs of co-ordinates:

```
10 LET n = 57200
20 INPUT "x =" ; x : INPUT "y = "; y
30 POKE n,y : POKE n + 1,x
40 LET n = n + 2
50 GOTO 20
```

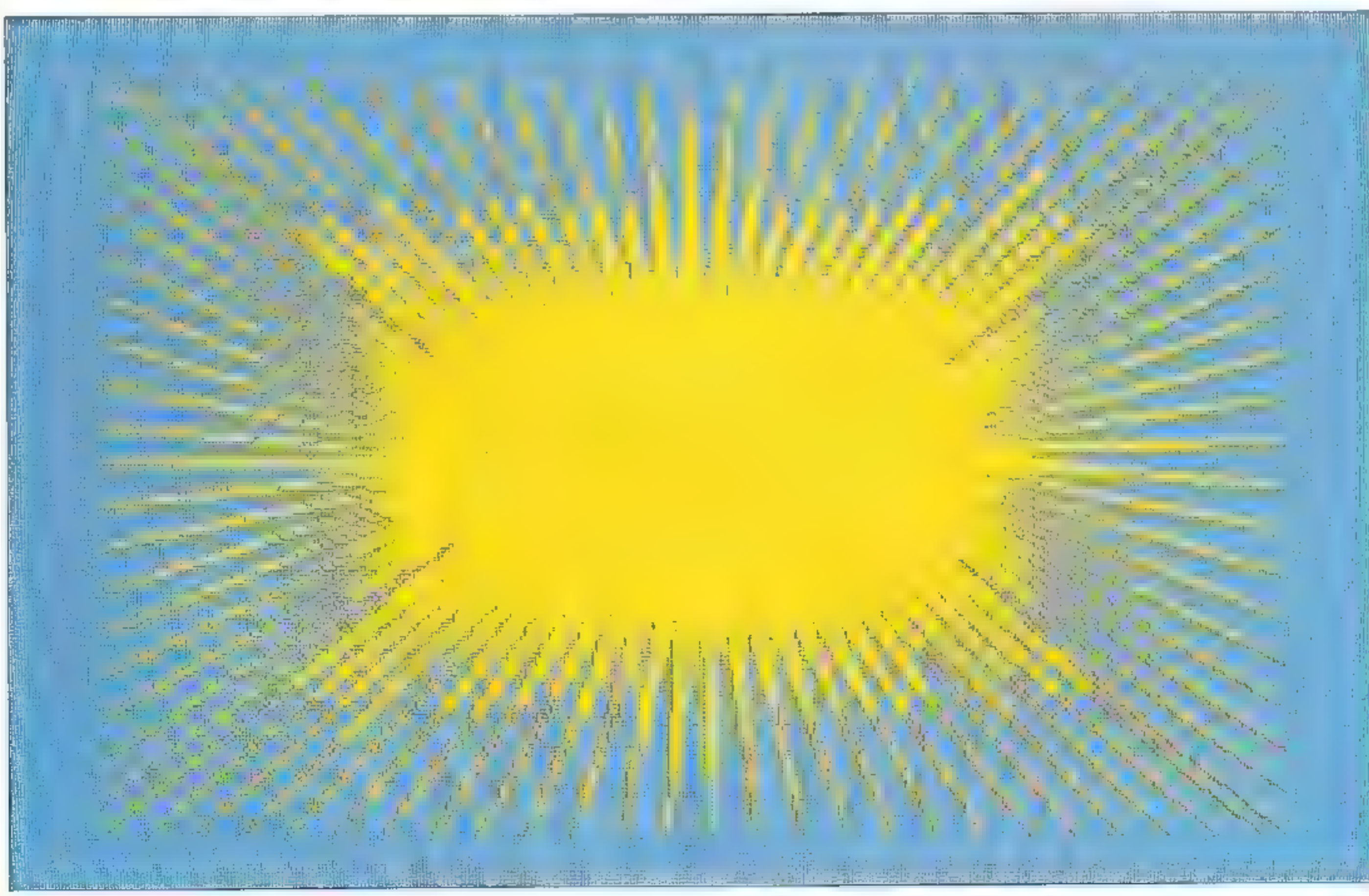
400 bytes from 57200 are reserved in memory for this purpose, so you can draw 199 lines with the routines.

These routines are especially useful for plotting the same shape on the screen repeatedly, since points once stored in memory can be called by the routine almost instantaneously.

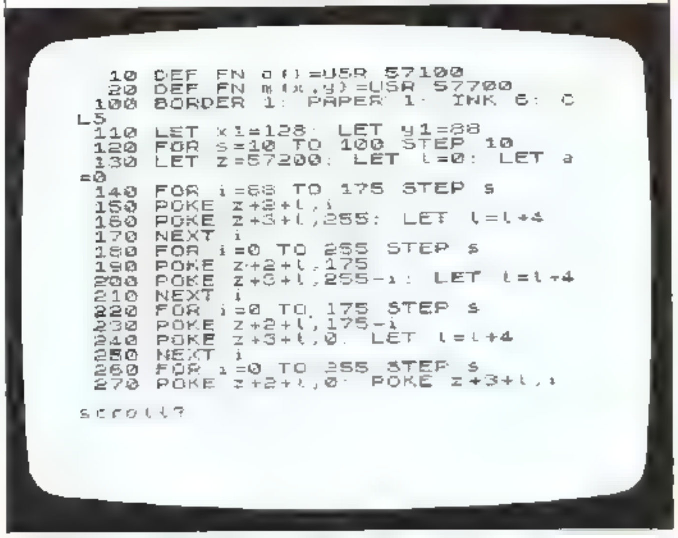
MULTILINE PROGRAM

00:22 **seconds**

How the program works An explosion effect is obtained by drawing a continuous line joining points on the edge of the screen with points on an ellipse. The shape is then filled. **Lines 140-330** POKE into memory the values of points around the edges of the screen. **Lines 350 to 400** POKE into memory points on an ellipse. **Lines 410 and 420** POKE values of 255 to complete the table of points.



MULTILINE PROGRAM

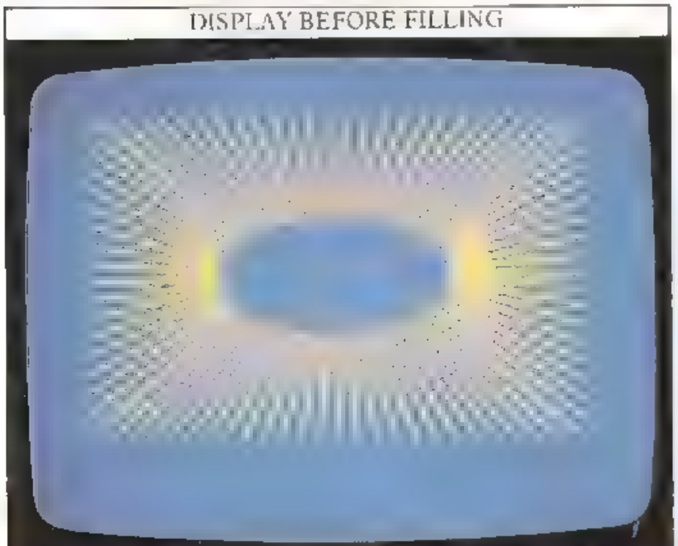


```
  10 DEF FN o()=USR 57100
  20 DEF FN m(x,y)=USR 57700
 100 BORDER 1: PAPER 1: INK 6: C
LS
 110 LET x1=128: LET y1=88
 120 FOR s=10 TO 100 STEP 10
 130 LET z=57200: LET l=0: LET a
=0
 140 FOR i=88 TO 175 STEP s
 150 POKE z+2+l,i
 160 POKE z+3+l,255: LET l=l+4
 170 NEXT i
 180 FOR i=0 TO 255 STEP s
 190 POKE z+2+l,175
 200 POKE z+3+l,255-i: LET l=l+4
 210 NEXT i
 220 FOR i=0 TO 175 STEP s
 230 POKE z+2+l,175-i
 240 POKE z+3+l,0: LET l=l+4
 250 NEXT i
 260 FOR i=0 TO 255 STEP s
 270 POKE z+2+l,0: POKE z+3+l,i

scroll?
```

```
 280 LET l=l+4
 290 NEXT i
 300 FOR i=0 TO 87 STEP s
 310 POKE z+2+l,i
 320 POKE z+3+l,255: LET l=l+4
 330 NEXT i
 340 LET ad=8*PI/l
 350 FOR i=0 TO l+4 STEP 4
 360 LET x=x1+INT (60*COS a)
 370 LET y=y1+INT (30*SIN a)
 380 POKE z+i,y: POKE z+i+1,x
 390 LET a=a+ad
 400 NEXT i
 410 POKE z+l+2,255
 420 POKE z+l+3,255
 430 RANDOMIZE FN o()
 440 PAUSE 100
 450 RANDOMIZE FN m(x1,y1)
 460 PAUSE 100: CLS
 470 NEXT s

0 OK, 0:1
```

The multiline program draws a long sequence of lines, which are then filled by the fill routine. Both routines must be in memory for the program to RUN.

DISPLAY BEFORE FILLING



FNo

MULTIPLE LINE-DRAW ROUTINE

Start address 57100 **Length** 40 bytes
Other routines called Line-draw routine (FNg).
What it does Draws a series of lines on the screen, from a specified list of co-ordinates.

Using the routine Co-ordinates of lines to be plotted are stored in a table at memory location 57200. Up to 200 lines can be stored in this area of memory. Points in the table must be specified by the y co-ordinate (0<=y<=175) followed by the x co-ordinate (o<=x<=255), rather than the other way round.
To stop the routine POKE specify a y co-ordinate of 255. The routine will continues plotting points until it reaches this y co-ordinate; if you omit the 255, the routine will continue to plot points using whatever numbers are in memory after the co-ordinate table.

ROUTINE LISTING

```
8100 LET b=57100: LET l=35: LET
z=0: RESTORE 8110
8101 FOR i=0 TO l-1: READ a
8102 POKE (b+i),a: LET z=z+a
8103 NEXT i
8104 LET z=INT (((z/l)-INT (z/l)
)*l)
8105 READ a: IF a<>z THEN PRINT
"??": STOP

8110 DATA 33,112,223,94,35
8111 DATA 86,237,83,26,237
8112 DATA 35,126,254,255,32
8113 DATA 1,201,95,35,86
8114 DATA 43,229,42,26,237
8115 DATA 205,51,237,225,24
8116 DATA 228,0,0,0,0
8117 DATA 17,0,0,0,0
```

FNp

MULTIPLE XOR-LINE ROUTINE

Start address 57000 **Length** 20 bytes
Other routines called Multiple line-draw routine (FNo).
What it does Draws a series of Exclusive/OR lines on the screen, using points specified in a table.

Using the routine This routine works in the same way as the multiple line-draw routine, but can be called twice with the same table of co-ordinates to erase the lines drawn. Remember as before to POKE points in the order y,x. Co-ordinates are stored in memory from location 57200, and the final point must be followed by a y co-ordinate of 255.

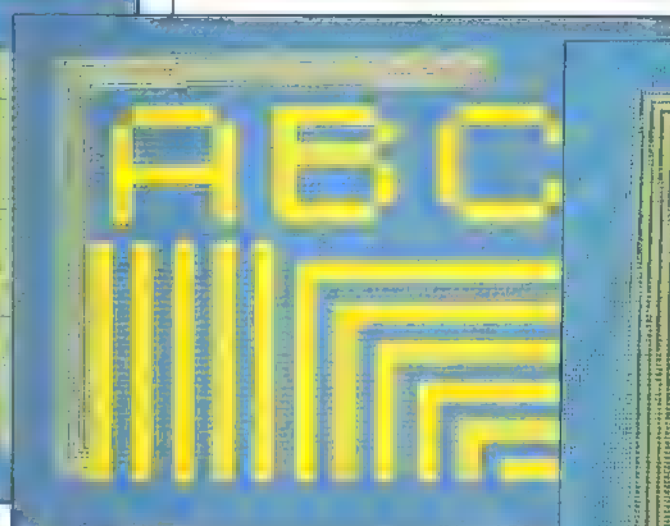
ROUTINE LISTING

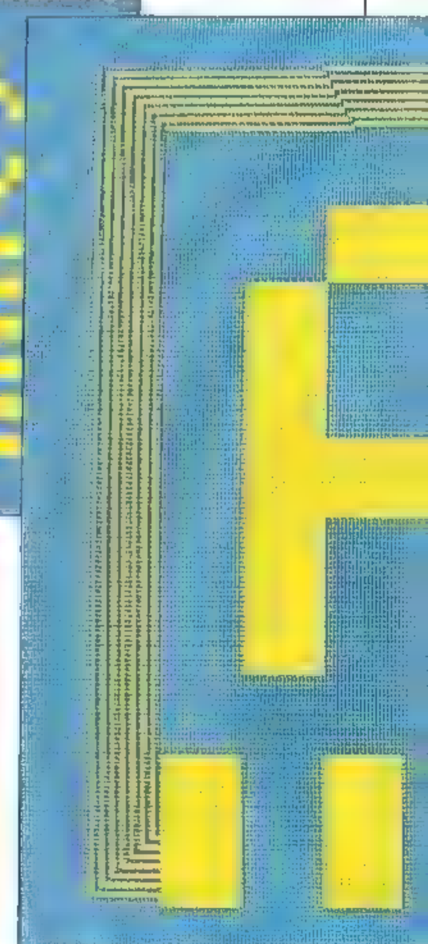
```
8150 LET b=57000: LET l=15: LET
z=0: RESTORE 8160
8151 FOR i=0 TO l-1: READ a
8152 POKE (b+i),a: LET z=z+a
8153 NEXT i
8154 LET z=INT (((z/l)-INT (z/l)
)*l)
8155 READ a: IF a<>z THEN PRINT
"??": STOP

8160 DATA 62,168,50,223,237
8161 DATA 205,12,223,62,176
8162 DATA 50,223,237,201,0
8163 DATA 13,0,0,0,0
```

MAGNIFICATION AND REDUCTION 1

One of the most dramatic uses for machine code is to magnify a portion of the Spectrum screen. The principle behind magnification is straightforward. To double the size of a single byte, for example 00110010 (a value of 50 in decimal), simply rotate it left one bit, thus making 01100100 (which is equivalent to 100 in decimal). Using this principle of doubling, you can magnify whole sections of a screen. The magnification routine, FNq, given here, is based on this idea. The routine simply requires you to specify the screen area to be enlarged.

The magnification routine is accompanied by a reduction routine, FNr, which is used to reduce an already-magnified area. The reduction routine actually forms part of the magnification routine, with the start address of the second routine being a call which "hooks" into the main routine. The reduction routine restores the screen as it was before the magnification, and works by the magnification routine saving the entire screen each time it is called before magnifying any area; the reduction routine simply displays this area from memory on the screen. Each time the magnification





routine is called, therefore, any former stage of magnification is deleted from memory, so the reduction routine can only be used to reduce a magnified area once.

The magnification program

This program uses the magnification routine to repeatedly enlarge a part of the screen. By adding the following lines:

```
30 DEF FNr()=USR 56957
280 RANDOMIZE FNr()
```

you can incorporate the reduction routine into the program. This will have the effect of reducing the enlarged area to its last state.

Remember that the magnification program calls the multiline routine, FNo, to draw the background pattern; this routine must also be present in memory for the program to RUN correctly.

MAGNIFICATION PROGRAM

```
  10 DEF FN o()=USR 57100
  20 DEF FN q(x,y,h,v)=USR 56700
 100 BORDER 1: PAPER 1: INK 6: C
LS
 110 LET x1=2: LET x2=174
 120 LET y1=174: LET y2=2
 130 FOR i=57200 TO 57598 STEP 8
 140 POKE i,y1: POKE i+1,x1
 150 POKE i+2,y2: POKE i+3,x1
 160 POKE i+4,y2: POKE i+5,x2
 170 POKE i+6,y1: POKE i+7,x2
 180 LET y1=y1-2: LET x1=x1+2
 190 LET y2=y2+2: LET x2=x2-2
 200 NEXT i
 210 POKE 57546,255
 220 RANDOMIZE FN o()
 230 INPUT a$: PRINT AT 2,2;a$
 240 FOR i=1 TO 5
 250 RANDOMIZE FN q(2,2,15,10)
 260 PAUSE 100
 270 NEXT i
 280 GO TO 230

0 OK, 0:1
```

MAGNIFICATION PROGRAM

00:05 seconds
(to magnify area)

How the program works
Lines 110-220 use the multiline routine to draw a series of lines.
Lines 130-200 POKE co-ordinates of the lines to be drawn.
Line 230 waits for text to be entered.
Lines 240-270 magnify the area with text five times.

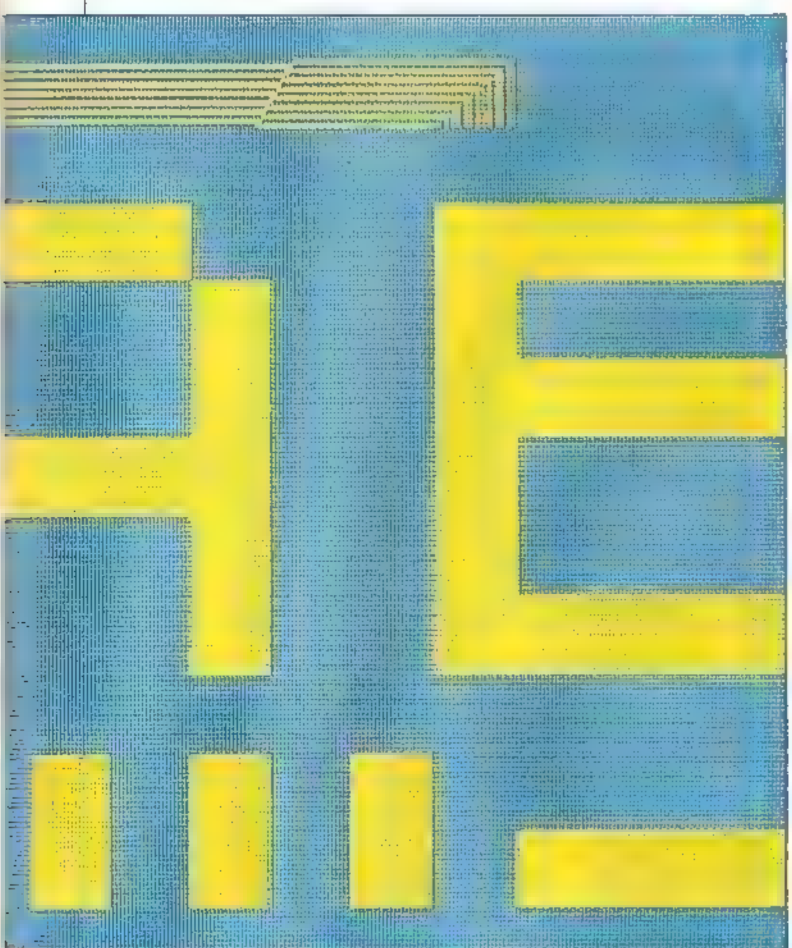
FNq

MAGNIFICATION ROUTINE

Start address 56700 **Length** 290 bytes
What it does Magnifies a specified screen area to double its previous size.

Using the routine The routine uses character co-ordinates, as in the window ink and paper routines (FNb and FNc), rather than pixel co-ordinates. Remember that these start from the top left-hand corner of the screen. The routine can be used to magnify the same area repeatedly, increasing the enlargement each time.
Since an area is doubled in size by the routine it is easy when magnifying to make part of the area disappear off the screen. To prevent crashes occurring, use the tests in the parameter table to make sure that the area when enlarged will not be off the screen. These tests can be incorporated into your programs.

ROUTINE PARAMETERS	
DEF FN q(x,y,h,v)	
x,y	specify top left-hand corner of area to be magnified (x<32, y<22)
h,v	specify horizontal and vertical sizes of area (x+(2*h)<32,y+(2*v)<22)



LISTING FOR BOTH ROUTINES

```
8200 LET b=56700: LET l=285: LET
 z=0: RESTORE 8210
8201 FOR i=0 TO l-1: READ a
8202 POKE (b+i),a: LET z=z+a
8203 NEXT i
8204 LET z=INT (((z/l)-INT (z/l)
)*l)
8205 READ a: IF a<>z THEN PRINT
"??": STOP

8210 DATA 42,11,92,1,4
8211 DATA 0,9,86,14,8
8212 DATA 9,94,237,83,137
8213 DATA 222,9,126,50,140
8214 DATA 222,9,126,50,139
8215 DATA 222,58,138,222,71
8216 DATA 58,140,222,128,230
8217 DATA 224,40,6,62,31
8218 DATA 144,50,140,222,58
8219 DATA 137,222,71,58,139

8220 DATA 222,128,214,22,[illegible]6
8221 DATA 6,62,21,144,50
8222 DATA 139,222,237,91,137
8223 DATA 222,123,230,24,246
8224 DATA 64,103,123,230,7
8225 DATA 183,31,31,31,31
8226 DATA 130,111,34,141,222
8227 DATA 17,0,64,167,237
8228 DATA 82,17,0,118,25
8229 DATA 34,143,222,205,113

8230 DATA 222,42,141,222,237
8231 DATA 91,143,222,58,139
8232 DATA 222,71,197,1,2
8233 DATA 4,197,205,20,222
8234 DATA 193,16,249,42,141
8235 DATA 222,205,79,222,34
8236 DATA 141,222,6,4,13
8237 DATA 32,235,237,91,143
8238 DATA 222,205,89,222,237
8239 DATA 83,143,222,193,16

8240 DATA 217,2[illegible]1,58,140,222
8241 DATA 71,34,145,222,237
8242 DATA 83,147,222,197,205
8243 DATA 54,222,193,16,249
8244 DATA 42,145,222,237,91
8245 DATA 147,222,229,205,99
8246 DATA 222,225,36,36,20
8247 DATA 201,26,1,2,4
8248 DATA 197,245,175,119,241
8249 DATA 23,245,203,22,241

8250 DATA 203,22,16,247,35
8251 DATA 193,13,32,237,19
8252 DATA 201,62,32,133,111
8253 DATA 208,62,8,132,103
8254 DATA 201,62,32,131,95
8255 DATA 208,62,8,130,87
8256 DATA 201,58,140,222,203
8257 DATA 39,71,126,36,119
8258 DATA 37,35,16,249,201
8259 DATA 33,0,64,17,0

8260 DATA 118,1,0,26,237
8261 DATA 176,201,33,0,118
8262 DATA 17,0,64,1,0
8263 DATA 26,237,176,201,2
8264 DATA 2,5,10,130,72
8265 DATA 226,118,98,78,194
8266 DATA 125,0,0,0,0
8267 DATA 38,0,0,0,0
```

FNr

REDUCTION ROUTINE

Start address 56957 **Length** 290 bytes
What it does Reduces a previously enlarged routine to its original size.

Using the routine Each time the magnification routine is called, it saves in memory the screen as it was before magnification. The reduction routine simply displays this saved screen. Thus the reduction routine cannot repeatedly reduce an area on the screen; it will only show whatever was on the screen before the last magnification.

MAGNIFICATION AND REDUCTION 2

The program on this page gives an indication of the capabilities of the magnification routine. From an initial display, the magnification routine is called three times to enlarge different areas of the screen. As a further sophistication, these enlarged areas are then coloured using the window paper routine, and a line is drawn around the edge of each area.

The various shapes are drawn by different subroutines at lines 300, 400, 500 and 600, and each shape is drawn higher up a column by increasing the y co-ordinate before calling the subroutine. After a single column has been completed with five shapes, the display is repeated by the loop beginning at line 110 which sets a new value for the x co-ordinate.

After the subroutines have been completed, line 700 uses the magnification routine for the first time. Each time an area is magnified, the box routine is then called to draw a black line around the edge of the enlarged area (in this case, line 710). After a pause, the new area is coloured, and another area is enlarged.

SWEETS PROGRAM

```
 10 DEF FN c(x,y,h,v,c,b,f)=USR
62600
 20 DEF FN g(x,y,p,q)=USR 60700
 30 DEF FN h(x,y,h,v)=USR 60400
 40 DEF FN j(x,y,r,s,f)=USR 589
00
 50 DEF FN m(x,y)=USR 57700
 60 DEF FN q(x,y,h,v)=USR 56700
100 BORDER 1: PAPER 6: INK 0: C
LS
110 FOR x=9 TO 209 STEP 40
120 LET x1=x: LET y1=10
130 GO SUB 500
140 LET y1=y1+35
150 GO SUB 600
160 LET y1=y1+30
170 GO SUB 400
180 LET y1=y1+30
190 GO SUB 300
200 LET y1=y1+40
210 GO SUB 400
220 NEXT x

scroll?
```

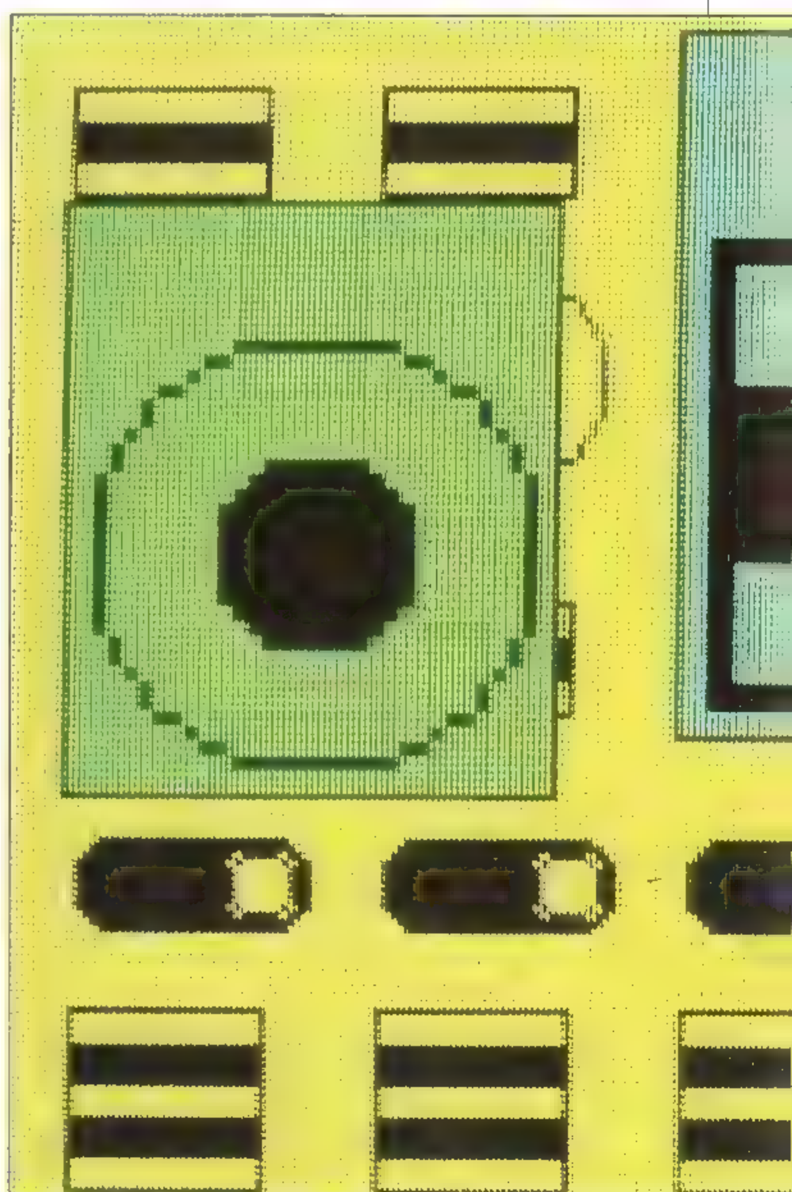
```
230 PAUSE 50
240 GO TO 700
250 STOP
300 RANDOMIZE FN j(x1+15,y1+15,
7,0,255)
310 RANDOMIZE FN m(x1+15,y1+15)
320 RANDOMIZE FN j(x1+15,y1+15,
15,0,255)
330 RETURN
400 RANDOMIZE FN h(x1,y1,25,15)
410 RANDOMIZE FN h(x1,y1+5,25,5
)
420 RANDOMIZE FN m(x1+1,y1+6)
430 RETURN
500 RANDOMIZE FN h(x1,y1,25,25)
510 RANDOMIZE FN h(x1,y1+5,25,5
)
520 RANDOMIZE FN m(x1+1,y1+6)
530 RANDOMIZE FN h(x1,y1+15,25,
5)
540 RANDOMIZE FN m(x1+1,y1+16)
550 RETURN

scroll?
```

Relocating areas in memory

The magnification routine works by storing in memory whatever is on the screen in the specified area. It uses 8K of memory, stored from location 30208. If your BASIC program is particularly long, you may find that this area is required by your program. By POKEing the following three numbers:

```
POKE 56793,176
POKE 56950,176
POKE 56959,176
```

you can place the code about 18K higher in memory.

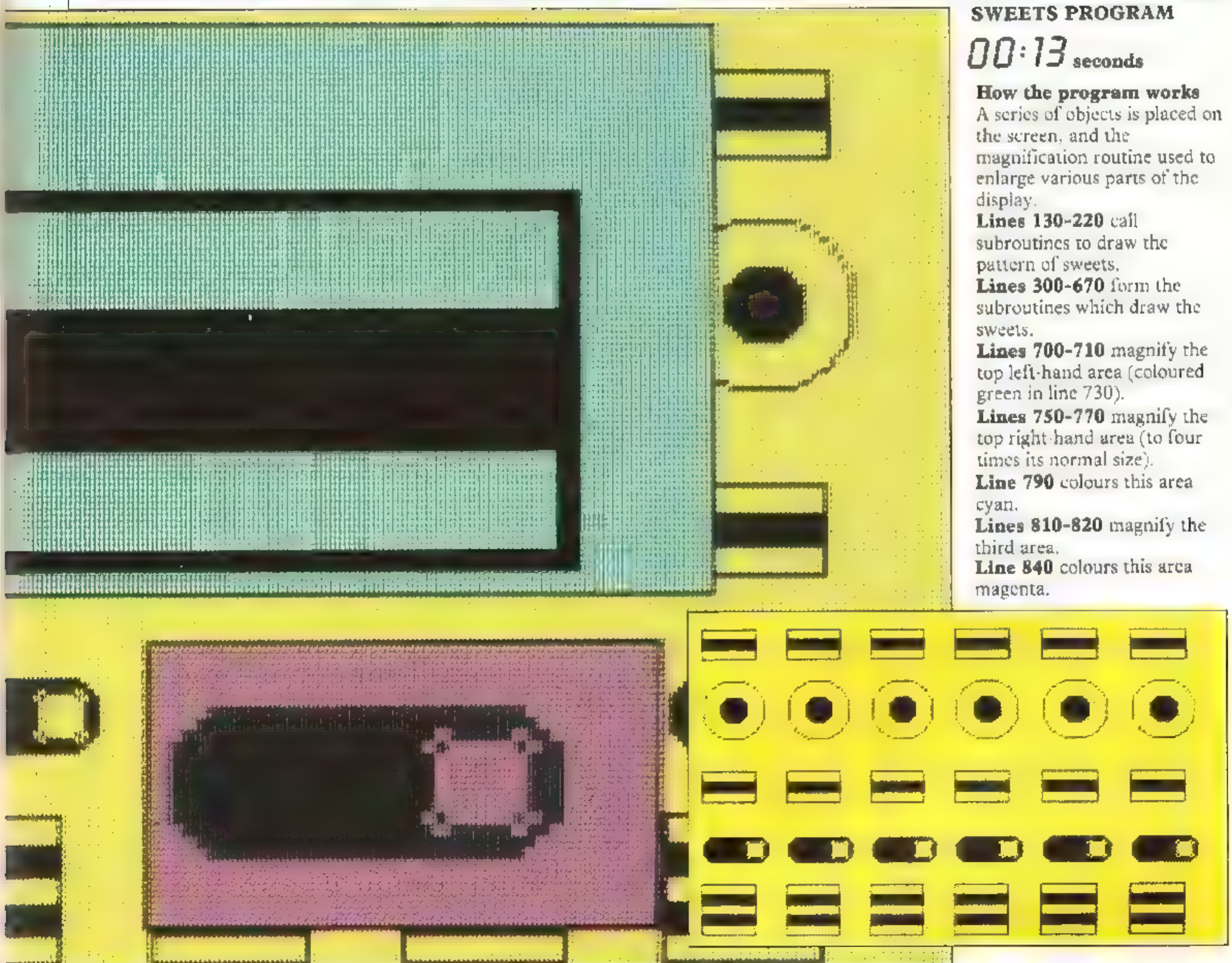


SWEETS PROGRAM CONTD.

```
 600 RANDOMIZE FN j(x1+25,y1+7,6
,0,255)
 610 RANDOMIZE FN j(x1+25,y1+7,7
,0,255)
 620 RANDOMIZE FN j(x1+25,y1+7,5
,0,255)
 630 RANDOMIZE FN g(x1+5,y1+1,x1
+21,y1+1)
 640 RANDOMIZE FN g(x1+5,y1+13,x
1+21,y1+13)
 650 RANDOMIZE FN j(x1+7,y1+7,7,
63,192)
 660 RANDOMIZE FN m(x1+7,y1+7)
 670 RETURN
 700 RANDOMIZE FN q(1,4,4,5)
 710 RANDOMIZE FN h(8,176-112,64
,80)
 720 PAUSE 100
 730 RANDOMIZE FN c(1,4,8,10,4,0
,0)
 740 PAUSE 100
 750 RANDOMIZE FN q(11,1,4,3)

scroll?
```

SWEETS PROGRAM CONTD.

```
 760 RANDOMIZE FN q(11,1,8,6)
 770 RANDOMIZE FN h(88,176-104,1
28,96)
 780 PAUSE 100
 790 RANDOMIZE FN c(11,1,16,12,5
,0,0)
 800 PAUSE 100
 810 RANDOMIZE FN q(16,14,5,3)
 820 RANDOMIZE FN h(128,176-160,
80,48)
 830 PAUSE 100
 840 RANDOMIZE FN c(16,14,10,6,3
,0,0)
 850 STOP




0 OK, 0:1
```



SWEETS PROGRAM

00:13 seconds

How the program works
A series of objects is placed on the screen, and the magnification routine used to enlarge various parts of the display.
Lines 130-220 call subroutines to draw the pattern of sweets.
Lines 300-670 form the subroutines which draw the sweets.
Lines 700-710 magnify the top left-hand area (coloured green in line 730).
Lines 750-770 magnify the top right-hand area (to four times its normal size).
Line 790 colours this area cyan.
Lines 810-820 magnify the third area.
Line 840 colours this area magenta.

SAVING AND LOADING DISPLAYS

The BASIC commands SAVE SCREEN$ and LOAD SCREEN$, used to save and load displays, have the disadvantage that they require nearly 8K of memory to save any display, no matter how simple. The screen compaction routine, FNs, allows you to store screens in a fraction of this space: the simpler the display, the less memory is required by the routine to store it. Even highly complex screen displays are stored in considerably less than 8K. As a guide, the three displays on this page require a total of just under 12K.

Previously saved displays can be displayed again using the decompaction routine, FNt. For both routines, the memory location of a display is specified by two parameters, containing the high and low bytes respectively of the start address.

COMPACTION PROGRAM

00:03 **seconds**

How the program works
Three screens are loaded using SCREEN$, compacted, and then displayed again in quick succession.
Line 110 sets values for high and low bytes of the address (30000) where the first display is to be stored.
Line 120 PRINTs this address.
Line 140 compacts the display.
Line 150 PEEKs start values for the next display.
Lines 160-200 repeat the operation for the other screens.
Lines 300-350 display the screens in turn.

COMPACTION DECOMPACTION PROGRAM

```
  10 DEF FN s(h,l)=USR 56600
  20 DEF FN t(h,l)=USR 56500
 100 LET a=56598: LET b=56599
 110 LET h1=117: LET l1=48
 120 LET ad=l1+(h1*256)
 130 PRINT ad: LOAD ""SCREEN$
 140 RANDOMIZE FN s(h1,l1)
 150 LET l2=PEEK a: LET h2=PEEK
b: LET ad=l2+(h2*256)
 160 PRINT ad: LOAD ""SCREEN$
 170 RANDOMIZE FN s(h2,l2)
 180 LET l3=PEEK a: LET h3=PEEK
b: LET ad=l3+(h3*256)
 190 PRINT ad: LOAD ""SCREEN$
 200 RANDOMIZE FN s(h3,l3)
 210 PAUSE 50
 300 RANDOMIZE FN t(h1,l1)
 310 PAUSE 50
 320 RANDOMIZE FN t(h2,l2)
 330 PAUSE 50
 340 RANDOMIZE FN t(h3,l3)
 350 PAUSE 50: GO TO 300

0 OK, 0:1
```

FNs

SCREEN COMPACTION ROUTINE

Start address 56600 **Length** 65 bytes
What it does Saves the current screen in a compacted form in memory.

Using the routine Parameters h and l are calculated by the formula

10 LET h=INT (store/256) : LET l=store−256*h

where "store" is the address in memory at which the screen is to be stored.
After storing a screen, you can find the start address for the next screen to be stored by PEEKing locations 23297 and 23296 (for h and l, the high and low bytes respectively of the address). This should be done immediately after compacting a display.

ROUTINE PARAMETERS	
DEF FN s(h,l)	
h,l	specify the high and low bytes of the data for the screen respectively (h,l<255)

ROUTINE LISTING

```
8300 LET b=56600: LET l=60: LET
z=0: RESTORE 8310
8301 FOR i=0 TO l-1: READ a
8302 POKE (b+i),a: LET z=z+a
8303 NEXT i
8304 LET z=INT (((z/l)-INT (z/l)
)*l)
8305 READ a: IF a<>z THEN PRINT
"??": STOP

8310 DATA 42,11,92,1,4
8311 DATA 0,9,86,14,8
8312 DATA 9,94,237,83,82
8313 DATA 221,33,0,64,6
8314 DATA 1,126,44,32,8
8315 DATA 36,245,124,254,91
8316 DATA 40,16,241,78,185
8317 DATA 32,4,4,32,238
8318 DATA 5,18,19,120,18
8319 DATA 19,24,227,241,18
8320 DATA 19,120,18,237,83
8321 DATA 22,221,201,0,0
8322 DATA 56,0,0,0,0
```

COMPACTION PROGRAM: SAMPLE DISPLAY 2

FNt

SCREEN DECOMPACTION ROUTINE

Start address 56500 **Length** 45 bytes
What it does Decompacts a screen previously stored at a specified address.

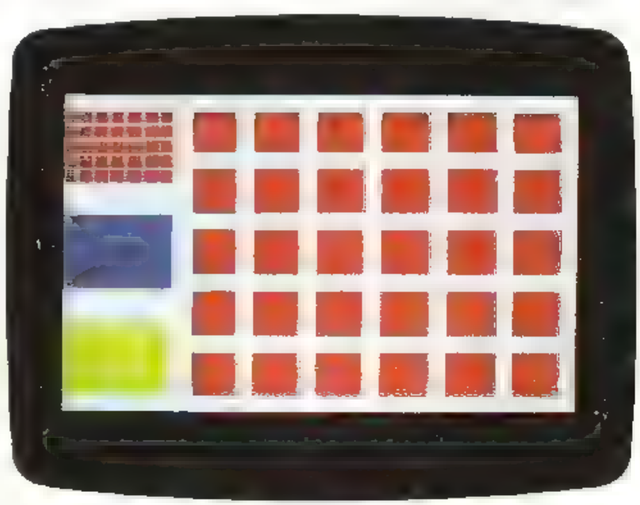
Using the routine This routine puts back onto the screen a routine previously stored by the compaction routine. The decompaction routine loads screens much more quickly than the LOAD SCREEN$ command.
To obtain the start addresses (h and l) of each screen compacted by the compaction routine (FNs), PEEK locations 23297 and 23296 (for h and l) after compacting a screen.

ROUTINE PARAMETERS	
DEF FN t(h,l)	
h,l	specify the high and low bytes of the data for the screen respectively (h,l<255)

ROUTINE LISTING

```
8350 LET b=56500: LET l=35: LET
z=0: RESTORE 8360
8351 FOR i=0 TO l-1: READ a
8352 POKE (b+i),a: LET z=z+a
8353 NEXT i
8354 LET z=INT (((z/l)-INT (z/l)
)*l)
8355 READ a: IF a<>z THEN PRINT
"??": STOP

8360 DATA 42,11,92,1,4
8361 DATA 0,9,86,14,8
8362 DATA 9,94,33,0,64
8363 DATA 26,245,19,26,19
8364 DATA 71,241,119,35,16
8365 DATA 252,124,254,91,32
8366 DATA 240,201,0,0,0
8367 DATA 23,0,0,0,0
```

COMPACTION PROGRAM: SAMPLE DISPLAY 3



The displays on this page were produced using the graphics editor, stored by the compaction program and then displayed in succession. The start addresses in memory (variables h,l) of any screens you draw will obviously differ from those given here.

ROUTINES CHECKLIST

The table shown below gives a summary of all the machine-code routines used in this book. This table does not explain every detail of using each routine; it is intended only as an aid when using the routines in your programs. If you have not used a routine before, you are recommended to read the introduction to the routine on the appropriate page of the book before using it in your program.

page	title	parameters		parameters	co-ordinates
11	partial screen clear	FNa(x,y,h,v)	x,y	start co-ordinates	character
			h,v	horiz. + vert. increment	character
11	window ink	FNb(x,y,h,v,c,b,f)	x,y	start co-ordinates	character
			h,v	horiz. + vert. increment	character
			c	colour	—
			b,f	BRIGHT or FLASH	—
13	window paper	FNc(x,y,h,v,c,b,f)	x,y	start co-ordinates	character
			h,v	horiz. + vert. increment	character
			c	colour	—
			b,f	BRIGHT or FLASH	—
15	enlarged horizontal text	FNd(x,y)	x,y	start co-ordinates	character
15	enlarged vertical text	FNe(x,y)	x,y	start co-ordinates	character
17	point-plot	FNf(x,y)	x,y	start co-ordinates	pixel
21	line-draw	FNg(x,y,p,q)	x,y	start co-ordinates	pixel
			p,q	end point co-ordinates	pixel
25	box-draw	FNh(x,y,h,v)	x,y	start co-ordinates	pixel
			h,v	horiz. + vert. increment	pixel
27	triangle	FNi(x,y,p,q,r,s)	x,y	start co-ordinates	pixel
			p,q	co-ordinates of second point	pixel
			r,s	co-ordinates of third point	pixel
29	squares table				
29	master curve				
31	arc	FNj(x,y,r,s,f)	x,y	start co-ordinates	pixel
			r	length of radius	pixel
			s,f	start and finish point of arc	—
33	sector	FNk(x,y,r,s,f)	x,y	start co-ordinates	pixel
			r	length of radius	pixel
			s,f	start and finish point of arc	—
33	segment	FNl(x,y,r,s,f)	x,y	start co-ordinates	pixel
			r	length of radius	pixel
			s,f	start and finish point of arc	—
35	fill	FNm(x,y)	x,y	start co-ordinates	character
39	XOR-line	FNn(x,y,p,q)	x,y	start co-ordinates	pixel
			p,q	co-ordinates of second point	pixel
53	multiple line-draw	FNo()			
53	multiple XOR-line draw	FNp()			
55	magnification	FNq(x,y,h,v)	x,y	start co-ordinates	character
			h,v	horiz. + vert. increment	character
55	reduction	FNr()			
59	compaction	FNs(h,l)	h,l	high and low bytes	—
59	decompaction	FNt(h,l)	h,l	high and low bytes	—

Before using a routine you must first define it in your program using DEF FN followed by the correct number of parameters. Parameters passed to machine-code routines must always be whole numbers; if a parameter value is calculated by your program, then put an INT statement in front of it to ensure a whole-number value is passed to the routine.

ranges	bytes	address	check
0-32 and 0-24 0-32 and 0-24	100	63000	82
0-32 and 0-24 0-32 and 0-24 0-7 0=off, 1=on	135	62800	53
0-32 and 0-24 0-32 and 0-24 0-7 0=off, 1=on	150	62600	19
0-32 and 0-24	220	62200	0
0-32 and 0-24	215	61900	125
0-255 and 0-176	65	61500	24
0-255 and 0-176 0-255 and 0-176	215	60700	192
0-255 and 0-176 0-255 and 0-176	110	60400	86
0-255 and 0-176 0-256 and 1-176 0-256 and 1-176	80	60300	68
	60	59600	3
	525	59000	234
0-255 and 0-176 0-255 0-255	45	58900	17
0-255 and 0-176 0-255 0-255	45	58800	35
0-255 and 0-176 0-255 0-255	30	58700	18
0-32 and 0-24	195	57700	57
0-255 and 0-176 0-256 and 1-176	20	57600	13
	40	57100	17
	20	57000	13
0-32 and 0-24 0-32 and 0-24	290	56700	38
		56957	
0-255	65	56600	56
0-255	40	56500	28

MEMORY MAP

This chart shows how the Spectrum memory is organised when all the routines are present in memory. RAMTOP should be set to 55500 using a CLEAR command.

Code	Title	Address
FNa	partial screen clear	63000
FNb	window ink	62800
FNc	window paper	62600
	100-byte buffer	62500
FNd	enlarged horizontal text	62200
FNe	enlarged vertical text	61900
	300-byte buffer	61600
FNf	point plot	61500
FNg	line draw	60700
FNh	box draw	60400
FNi	triangle draw	60300
	600-byte buffer	59700
	squares table	59600
	master curve	59000
FNj	arc	58900
FNk	sector	58800
FNl	section	58700
FNm	fill	57700
FNn	exclusive/OR-line draw	57600
	400-byte buffer	57200
FNo	multiple line draw	57100
FNp	multiple XOR-line draw	57000
FNq	magnification	56700
FNr	reduction	56957
FNs	compaction	56600
FNt	decompaction	56500

ERROR TRAPPING

Error trapping in BASIC is carried out when an error message is displayed to show a mistake has occurred. This message is produced by a routine in the Spectrum ROM which prints on the screen the nature of the error.

When using machine code, however, it is often difficult to place restrictions on the way the routines are used. In most cases a determined user will be able to make the routine crash simply by passing it information which it does not understand. This could be checked within the machine-code routine itself to ensure that whatever is inputted is within the possible ranges you can type in, but to do this for all the routines in this book would require each routine to be perhaps doubled in length to incorporate the error checking required.

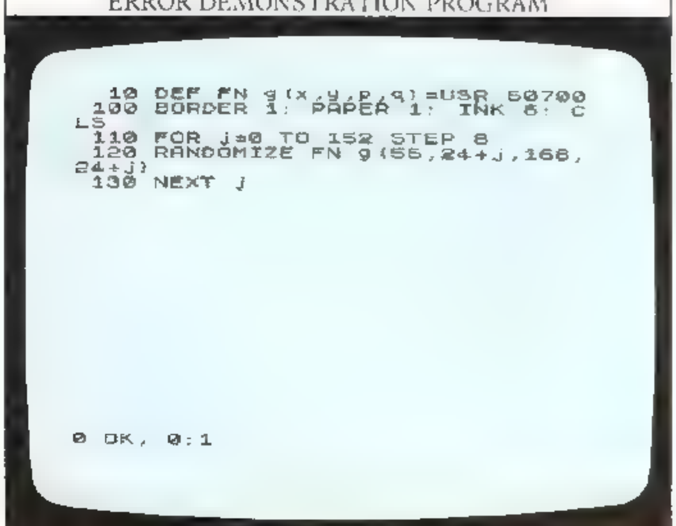
Preventing likely errors

In some cases it is quite easy, as well as helpful for the user, to add at least some error checking. A check routine has been added to the point-plot routine, for example, which means that although you may get rather unexpected results when you plot off-screen points, the routine is unlikely to crash. Try plotting a point which is off the screen and you will see the effects — a point will appear, but since the point specified is off the screen the routine will try to make sense of the data and plot a point at a position on the screen.

Error-trapping with the line-draw routine

Another part of the machine code which contains some error checking is the line-draw routine, as you can see from the error demonstration program shown here. Specifying lines off the top or bottom of the screen will result in "Integer out of range" being displayed, as happens when the program below is RUN with the co-ordinates shown. Errors in horizontal co-ordinates are more difficult to trap, since these co-ordinates lie between 0 and 255, the range of numbers that can be contained in one byte. If you use a number larger than 255 it is likely to be treated as if it were 255 less than its actual value, with the result that the line wraps round to the other side of the screen. This effect can be seen in the screen below, the result of specifying parameter values which are off the screen horizontally.

ERROR DEMONSTRATION PROGRAM



PLOTTING OFF THE SCREEN VERTICALLY

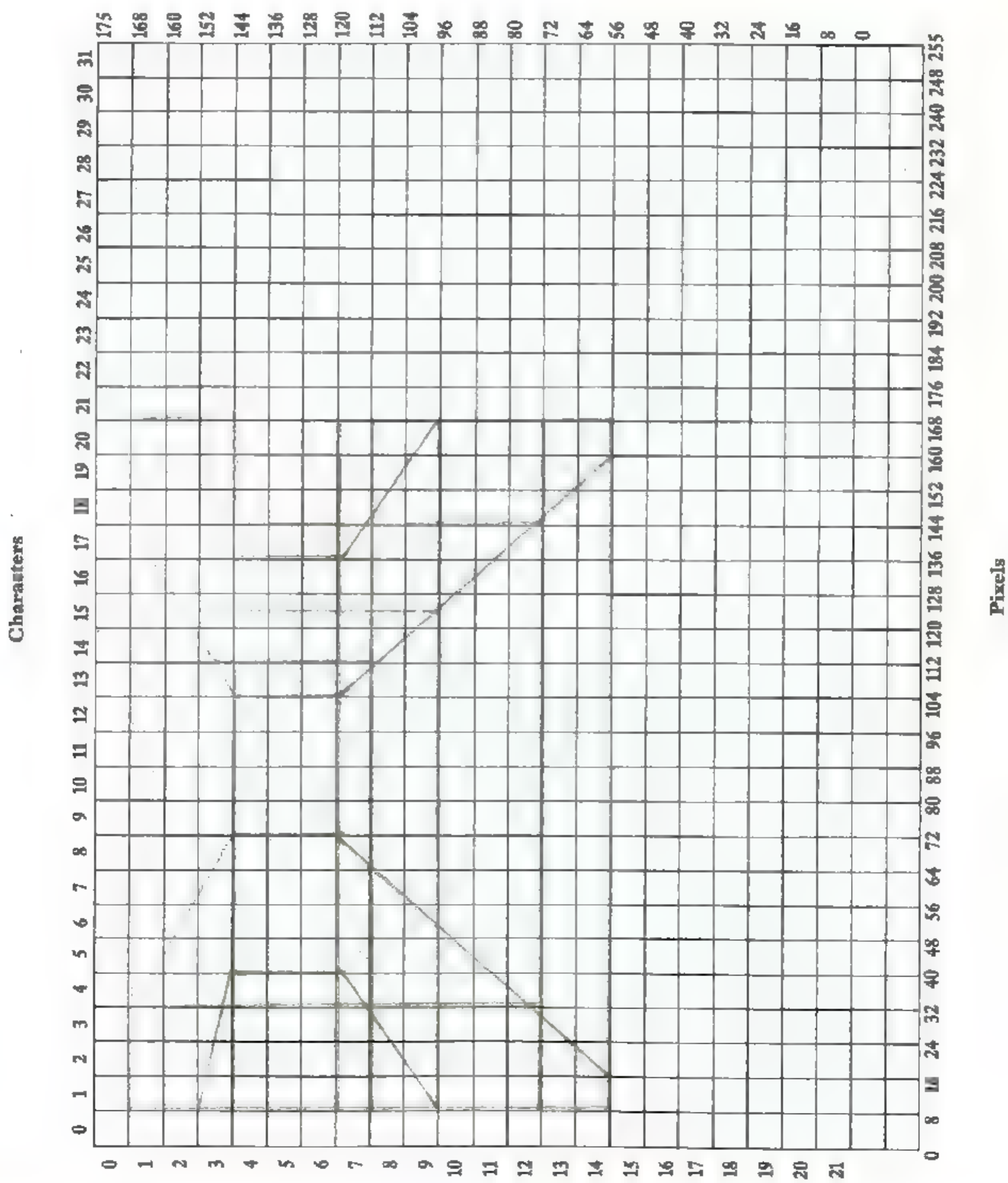
PLOTTING OFF THE SCREEN HORIZONTALLY

A similar effect can be seen with the curve routines, which cause odd effects when they go off the sides of the screen, but which will work within limits off the top and bottom of the screen.

The error trapping in these routines covers only the most likely errors you may make when using the machine code. As far as possible, you should keep to the limits and parameters specified for each routine.

GRAPHICS GRIDS

This grid shows screen divisions for both pixel and character co-ordinates. Points on the screen are defined by co-ordinates x (horizontal) and y (vertical). Character co-ordinates are measured from the top left-hand corner of the screen across and down. Pixel co-ordinates are measured from the bottom left-hand corner across and upwards. Pixel co-ordinates do not cover the bottom two lines of the screen.



INDEX

Main entries are given in **bold** type

Acknowledgments

A number of people helped and encouraged me with this book. Thanks to Alan and Michael at Dorling Kindersley, to Jacqui Lyons for her representation and to Andy Werbinski for reluctant assistance. I am particularly grateful, as always, to my parents, and to Martine.

Piers Letcher
Spring 1985

DK

Screen Shot

The bestselling teach-yourself programming course now takes you beyond BASIC to the world of advanced machine-code graphics.

Using a combination of simple BASIC programming and a collection of tailor-made, ready-to-run machine-code routines, this book shows you how to produce precision, high-resolution graphics in a fraction of the time they would take in BASIC alone. A keyboard-driven graphics editor and a wide variety of demonstration programs will help you open up the full potential of the ZX Spectrum – without the need for any knowledge of machine-code programming.

Together, Books Three and Four in this series form a complete, self-contained graphics system for Spectrum-owners.

All the programs in this book run on both 48K ZX Spectrum and ZX Spectrum+ machines.

“Far better than anything else reviewed on these pages . . . Outstandingly good”
BIG K

“As good as anything else that is available, and far better than most”
COMPUTING TODAY

“Excellent . . . As a series they could form the best ‘basic introduction’ to programming I've seen”
POPULAR COMPUTING WEEKLY

DORLING KINDERSLEY

ISBN 0-86318-103-1

£5.95

9 780863 181030

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM

IAN GRAHAM

BOOK ONE

A must for every beginner – the first full-colour guide to easy programming

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM

THE DK SCREEN-SHOT PROGRAMMING SERIES

Never has there been a more urgent need for a series of well-produced, straightforward, practical guides to learning to use a computer. It is in response to this demand that The DK Screen-Shot Programming Series has been created. It is a completely new concept in the field of teach-yourself computing. And it is the first comprehensive library of highly illustrated, machine-specific, step-by-step programming manuals.

BOOKS ABOUT THE ZX SPECTRUM

This is Book One in a series of unique step-by-step guides to programming the ZX Spectrum. Together with its companion volumes, it will build up into a self-contained teaching course that begins with the basic principles of programming, and progresses – via more sophisticated techniques and routines – to an advanced level.

BOOKS ABOUT OTHER COMPUTERS

Additional titles in the series will cover each of the world's most popular computers. These will include:

Step-by-Step Programming for the **BBC Micro**

Step-by-Step Programming for the **Commodore 64**

Step-by-Step Programming for the **Acorn Electron**

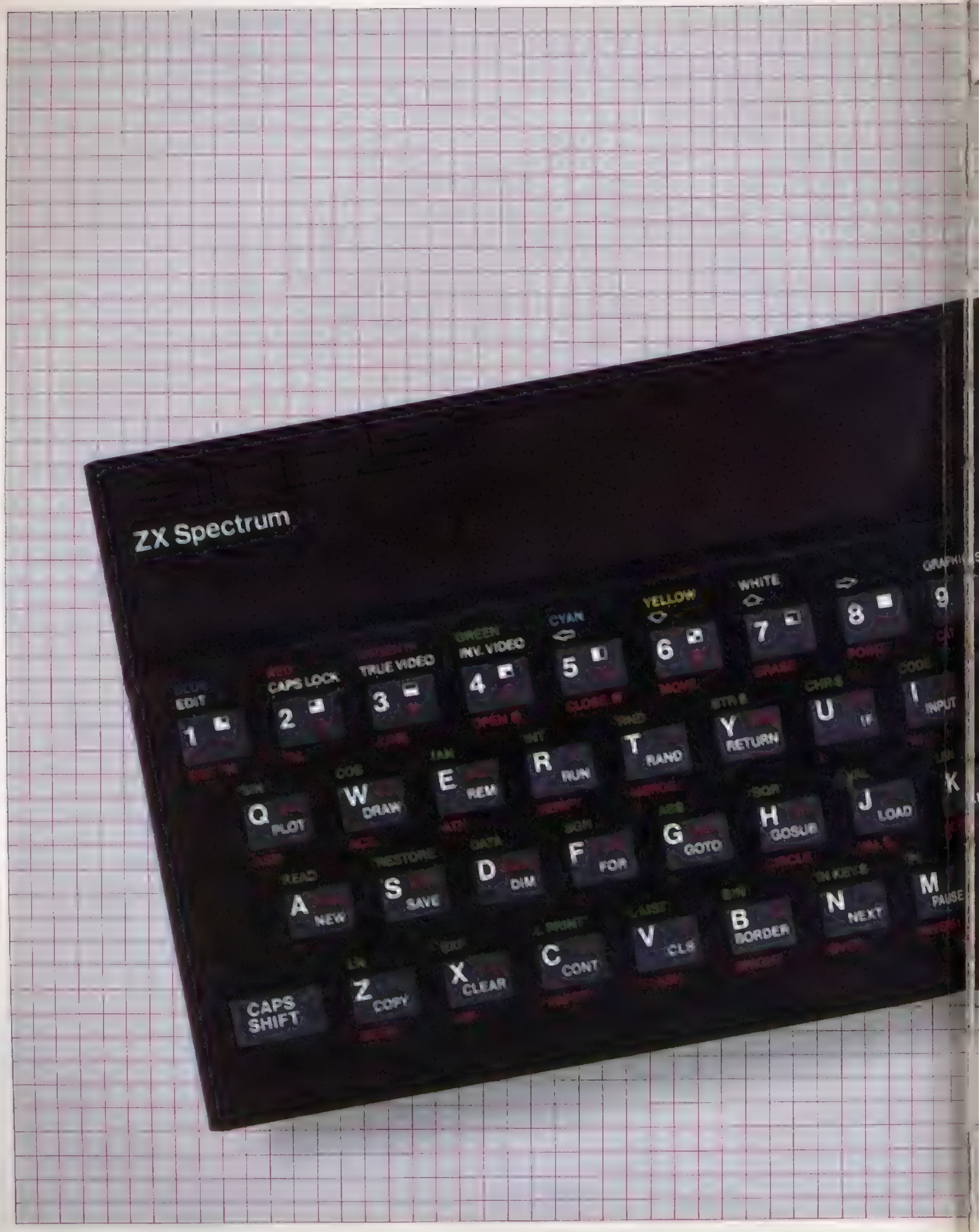
Step-by-Step Programming for the **Apple II**

Step-by-Step Programming for the **IBM PCjr**

IAN GRAHAM

After taking a B.Sc. in Applied Physics and a postgraduate diploma in Journalism at The City University, London, Ian Graham worked as assistant editor of *Electronics Today International* and deputy editor of *Which Video?* Since becoming a full-time freelance writer in 1982, he has contributed to a wide range of technical magazines (including *Computing Today, Video Today, Video Search, Hobby Electronics, Electronic Insight, Popular Hi-Fi, Science Now,* and *Next...*) and has also written a number of popular books on computers and computing. These include *Computer & Video Games, Information Technology, The Inside Story – Computers,* and *The Personal Computer Handbook* (co-written with Helen Varley).

BOOK ONE


ZX Spectrum
EDIT
1
CAPS LOCK
2
TRUE VIDEO
3
INV. VIDEO
4
GREEN
CYAN
5
YELLOW
6
WHITE
7
8
GRAPHICS
9
Q
PLOT
W
DRAW
E
REM
R
RUN
T
RAND
Y
RETURN
U
INPUT
A
NEW
S
SAVE
D
DIM
F
FOR
G
GOTO
H
GOSUB
J
LOAD
K
READ
RESTORE
DATA
CAPS
SHIFT
Z
COPY
X
CLEAR
C
CONT
V
CLS
B
BORDER
N
NEXT
M
PAUSE

DK

ScreenShot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM

IAN GRAHAM

DORLING KINDERSLEY·LONDON

BOOK ONE

CONTENTS

The DK Screen-Shot Programming Series was conceived, edited and designed by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

Project Editor David Burnie
Art Editor Peter Luff
Design Assistant Steve Wilson
Photography Vincent Oliver
Managing Editor Alan Buckingham
Art Director Stuart Jackman

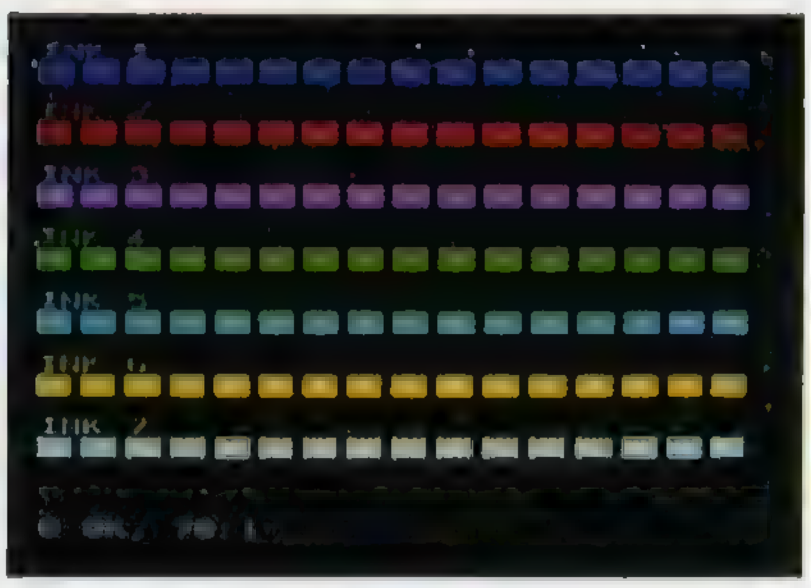
First published in Great Britain in 1984 by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.
Second impression 1984

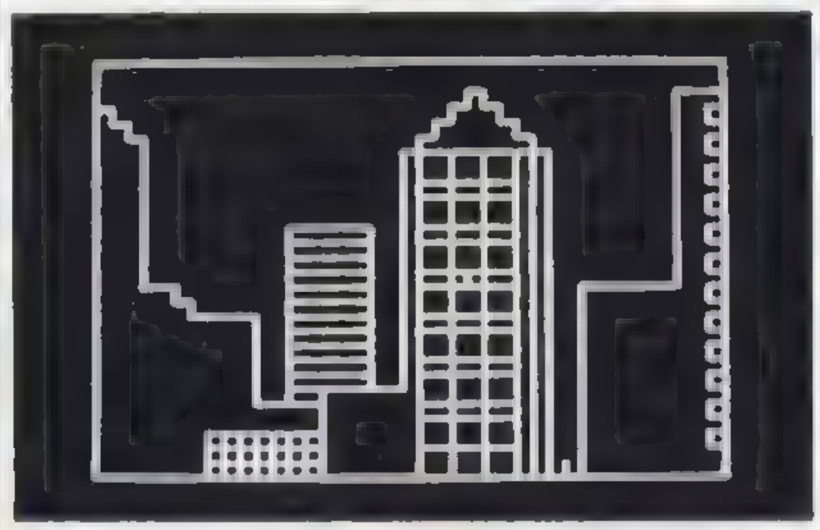
British Library Cataloguing in Publication Data

Graham, Ian, 1953–
Step by step programming for the ZX Spectrum. Book 1.
1. Sinclair ZX Spectrum (Computer)——Programming
I. Title
001.64'2 QA76.8.S625

ISBN 0-86318-026-4

Typesetting by The Letter Box Company (Woking) Limited, Woking, Surrey, England
Reproduction by Reprocolor Llovet S.A., Barcelona, Spain
Printed and bound in Italy by A. Mondadori, Verona





THE ZX SPECTRUM

Since its launch in April 1982, the ZX Spectrum has become one of the most popular home computers available today. Despite its small size, it is potentially very powerful. It contains the same basic components as much larger computers and uses its own version or "dialect" of the most popular home computer language, BASIC. The Spectrum offers an inexpensive way of learning to program a computer with many of the features found in larger, more elaborate systems.

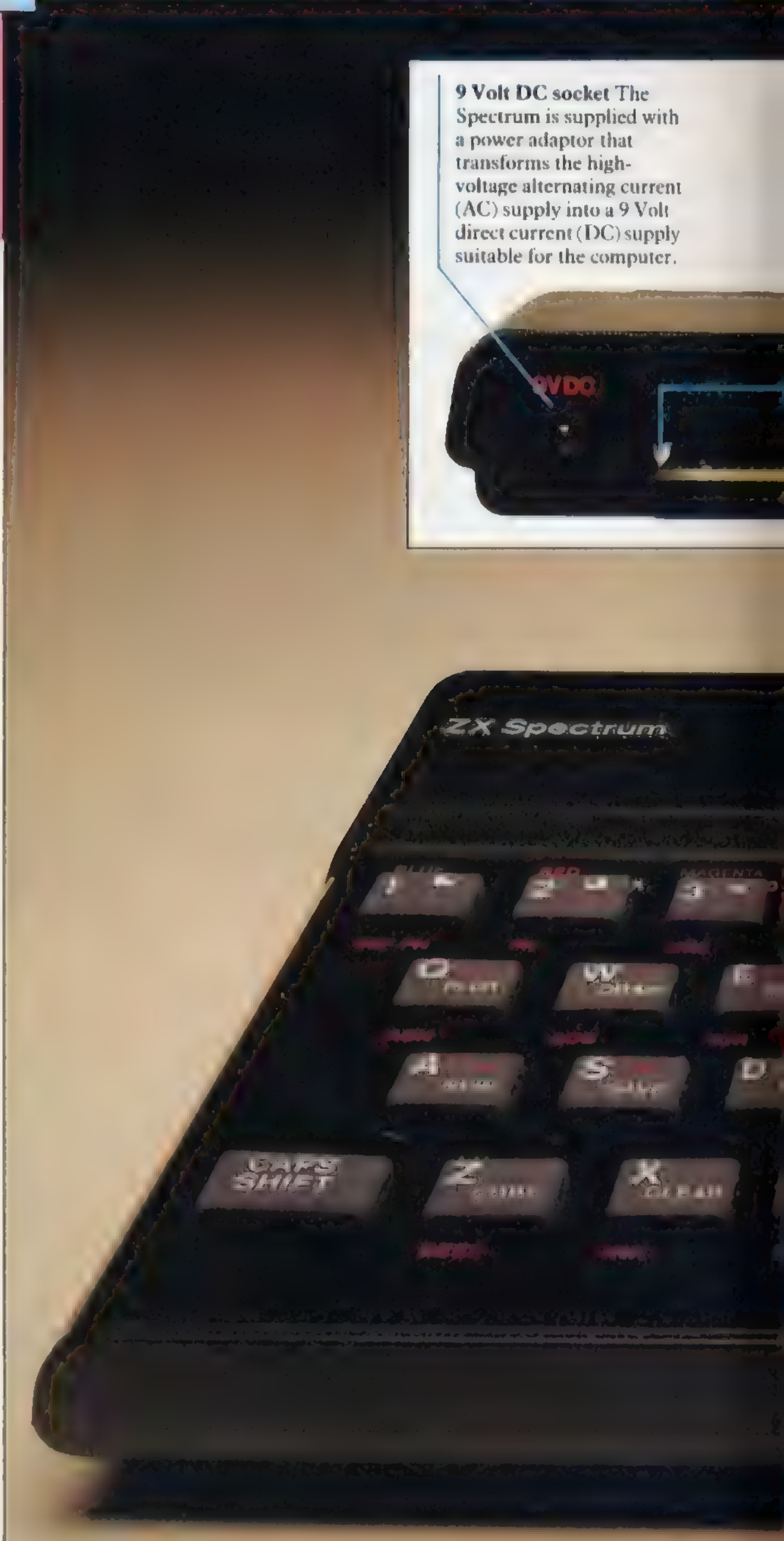
Two versions of the Spectrum are available. They are identical but for the size of the memory – the 48K model can hold more information than the 16K version. The 48K model has a memory capacity which is greater than many more expensive personal computers. All the programs in this book will work equally well on either the 16K or 48K machine.

Connectors and peripherals

From the outside the Spectrum seems to be little more than a slim black case with a keyboard on top, which is composed of soft moving keys. Although it looks like a typewriter keyboard, it is actually very different. As you can see on pages 10–11, each key can produce whole words as well as letters, and when used in programming, each key can perform up to six different functions.

If you turn the computer around so that you are looking at its back panel, you will see that there are four sockets and a slot. The small sockets connect the Spectrum to a television, cassette recorder and power supply. You can find out how to use a cassette recorder with the computer on page 60. The longer slot is the edge connector, which is actually a part of the Spectrum's circuit board exposed at the edge of the computer casing. The metallic strips on the edge of the board are used to connect extra pieces of hardware, such as printers, microdrives or joysticks, to the computer. When you are handling the computer, do not touch any of these contacts, as grease and dirt can cause mulfunctions if you later use the edge connector with any "peripherals".

The Spectrum produces a colour television picture, but it can also be viewed in black and white. In this case, the different colours show up as shades of grey. The computer itself produces all the sound effects used in programs. If you turn the Spectrum over so that you are looking at the bottom panel, you will be able to see the circle of holes in one corner that is the sound outlet for the Spectrum's tiny loudspeaker. It is capable of beeping or playing simple tunes.



Edge connector This multi-terminal socket connects the computer with a range of hardware including the Spectrum printer, Microdrive units and "analogue" controls such as games joysticks.

MIC socket When this is connected to a cassette recorder's microphone socket, programs can be transferred from the computer to be stored on cassette.

EAR socket This socket allows the Spectrum to receive stored programs from a tape cassette. It is connected to the cassette recorder's EAR socket.

TV socket The picture signals that the Spectrum produces are fed into a television's aerial socket from the Spectrum's TV socket, using a cable supplied with the computer.

INSIDE THE COMPUTER

The ZX Spectrum is constructed on a single printed circuit board. The major components are integrated circuits, or "chips", which look like thin rectangular slices of black plastic with up to 20 metal pins protruding from each edge. These connect the chips to contacts which run over the surface of the board.

At the heart of the Spectrum is a microprocessor which makes up the computer's Central Processing Unit (CPU). The CPU does all the computer's calculations, monitors the keyboard and acts on any key-presses it detects. But despite the CPU's complexity, it can only follow instructions that it is given. The instructions you type on the keyboard must first be translated into the numerical machine code that the computer works in.

The program required to perform this translation is stored permanently in a Read Only Memory (ROM), also known as a "non-volatile" memory. This memory is not free for you to store your own programs in – its contents can only be read. The program is unaffected by the computer being turned on or off.

The Spectrum's Random Access Memory (RAM) offers storage space for the user. Everything you type in on the keyboard is stored in RAM until you either type in the keyword NEW or switch off the power. Because the contents of RAM are erased when the power is interrupted, it is also called a "volatile" memory.

Main board components

The two versions of the Spectrum are distinguished by the size of their RAMs – 16K and 48K. The K stands for kilobytes, each equivalent to 1024 bytes. A byte is a standard piece of electronic information in binary form. It is made up of eight bits – pulses of electricity which each represent a 0 or 1. Bytes can be thought of as acting like words in the computer system, with bits being the letters. The main difference between the Spectrum's system of communication and English is that computer words are all eight letters long, and are made up from an alphabet containing only two letters. The Spectrum does everything from producing colour television pictures to playing tunes by using this binary code.

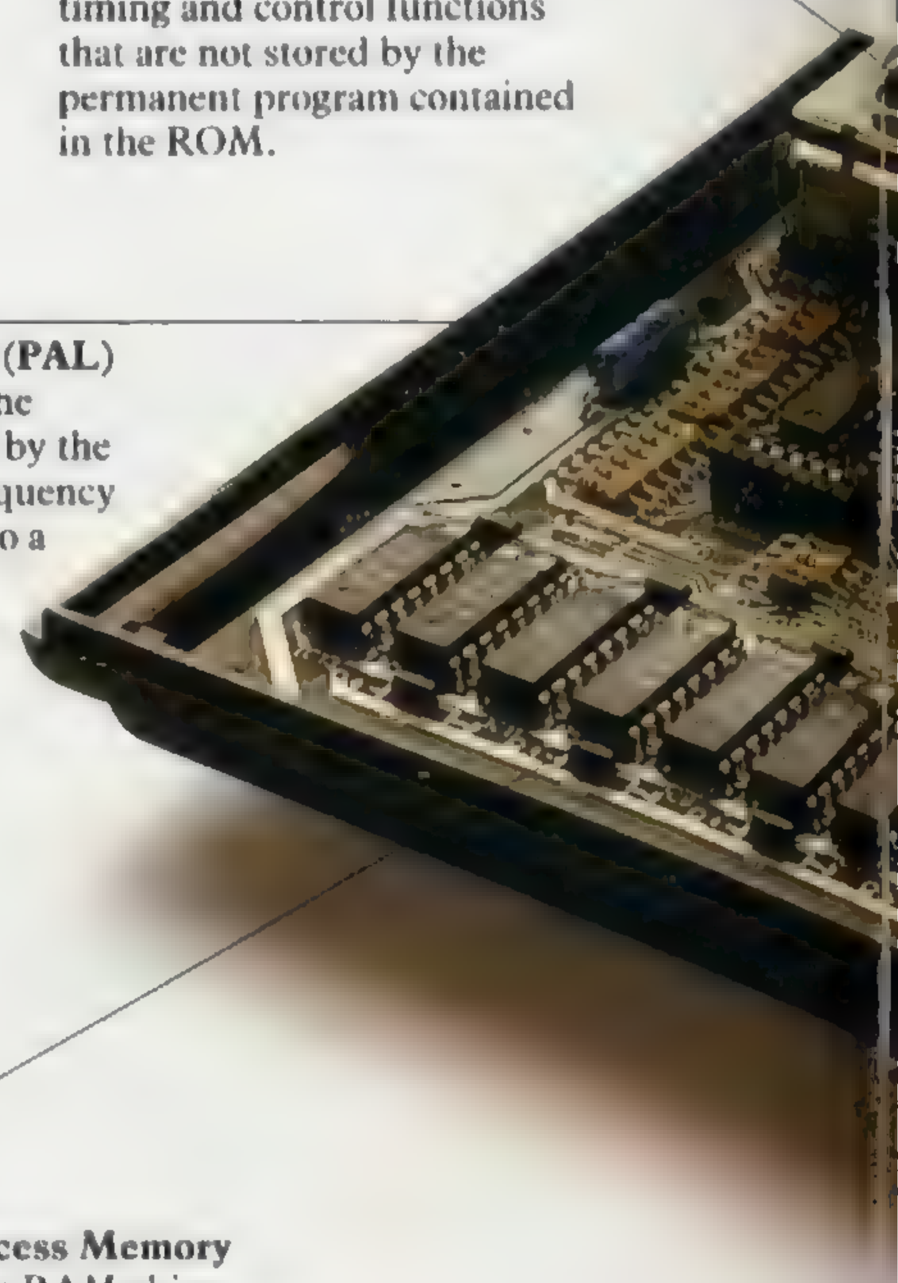
All the computer's activities are synchronized so that the correct information is available in the right place at the right time. This synchronization is achieved by an internal clock. The clock is based on a crystal oscillator that "ticks" at the rate of 3.5 million pulses per second. Additional timing and control operations are provided by a large chip called an Uncommitted Logic Array (ULA), which carries out complicated logic functions.

The microchip command chain All the chips within the Spectrum form an electronic chain of command, with the CPU performing all the executive tasks. The rest of the chips – including the RAMs, ROM and ULA – act as temporary or permanent information storage systems. These supply the CPU with the instructions it needs to carry out the computer's functions.

In this view of the computer's interior, the connectors that link the keyboard to the rest of the computer have been detached. It is advisable not to open your Spectrum as these connectors can easily be broken.

Uncommitted Logic Array (ULA) This provides additional timing and control functions that are not stored by the permanent program contained in the ROM.

Phase Alternation Line (PAL) encoder This converts the stream of data produced by the Spectrum into a high-frequency signal that can be fed into a television.



Random Access Memory (RAM) Eight RAM chips provide 16K of storage for all the programming information that the computer is given after being switched on. The second group of eight further RAM chips provides an additional 32K of memory in the 48K version of the Spectrum.


Cassette recorder sockets These are used to record or play back programs.
Logic chips This area deals with a variety of logic functions including those needed to "interface" with hardware linked to the computer.
Central Processing Unit (CPU) The executive part of the computer. This microprocessor carries out all calculations and controls activities in the rest of the computer, drawing on information held in both ROM and RAM.
Read Only Memory (ROM) This chip contains the instructions necessary to turn programs into a form that the computer's most important chip, the CPU, can understand. Unlike RAM, its contents do not disappear when the power is switched off. It contains the computer's BASIC interpreter.
9 Volt DC socket The power supply socket.
Voltage regulator This prevents changes in voltage disrupting the activities of the computer.
Loudspeaker The speaker produces notes and sound effects when called on by a program.

THE SPECTRUM KEYBOARD

On a conventional typewriter keyboard, each key is used to print a lower case letter and (with the shift key) the upper case version of the same letter. The Spectrum keyboard works like this, but because it has more than one shift key, it is much more versatile. The keys on the Spectrum are capable of selecting as many as six different functions. The 40 Spectrum keys can produce a total of around 200 letters, words and symbols. You may find using the keyboard a slow process to begin with, but once you have mastered the use of the shift keys, finding your way around all the different words and symbols will soon become second nature. Two different kinds of shift key – CAPS SHIFT and SYMBOL SHIFT – are used either independently or together to select key functions.

Keyboard technique

The Spectrum keys are moving, calculator-style buttons. Every time one is pressed, making a character or word appear on the television screen, the computer produces a single click to let you know that the contact has been made. If you hold a key down for more than a couple of seconds, the symbol is repeatedly printed.

You don't have to be an accomplished typist to use the keyboard. The Spectrum saves you a lot of typing because the command words used by the computer need not be typed out in full. Pressing a key once makes the whole word printed on the key appear on the television screen. To help you further, commands and symbols that are often used together in programs are, wherever possible, grouped on adjacent keys.

Understanding the cursor

The Spectrum accepts instructions only if they are in a logical sequence, and it actually tells you what kind of instruction or symbol it expects by showing one of five flashing cursors. If the cursor is a "K", the computer is expecting you to supply a keyword next. This is simply any of the command words in white on the keyboard. So, if you press the P key, the word PRINT, and not the letter P, appears on the screen. If the cursor is flashing "L", pressing the P key produces a lower case letter p on the screen. To produce a capital P, press the CAPS SHIFT key while pressing p. This will make the cursor change briefly from "L" to "C" to show that capitals have been selected. The "G" cursor will appear when you use the GRAPHICS key, while the "E" (extended mode) cursor will appear when you use the shift keys to select keywords printed on the keyboard in red or green. It reverts to "L" after ENTER is pressed.

Number keys These offer a quick and easy way of producing graphics when they are used after pressing CAPS SHIFT with GRAPHICS. The graphics symbol on the number key will then appear on the screen. Keys 0 to 7 also control the colours produced on the screen.

EDIT This is used to "extract" a line from a program in order to change or edit it. The EDIT function is selected by the CAPS SHIFT key.

BEEP When the shift keys are used to select the extended mode, this key programs the command which controls the Spectrum's sound synthesizer.

CAPS SHIFT This allows you to select the upper case (capital) version of a letter, instead of the lower case version normally used. It is also used with SYMBOL SHIFT to obtain the words and characters above and below the keys.

Screen display keys The red keywords under keys X to M produce the commands controlling the way the text and screen background is displayed. The keywords INK and PAPER, together with the white keyword BORDER, are used in conjunction with the colour keys.

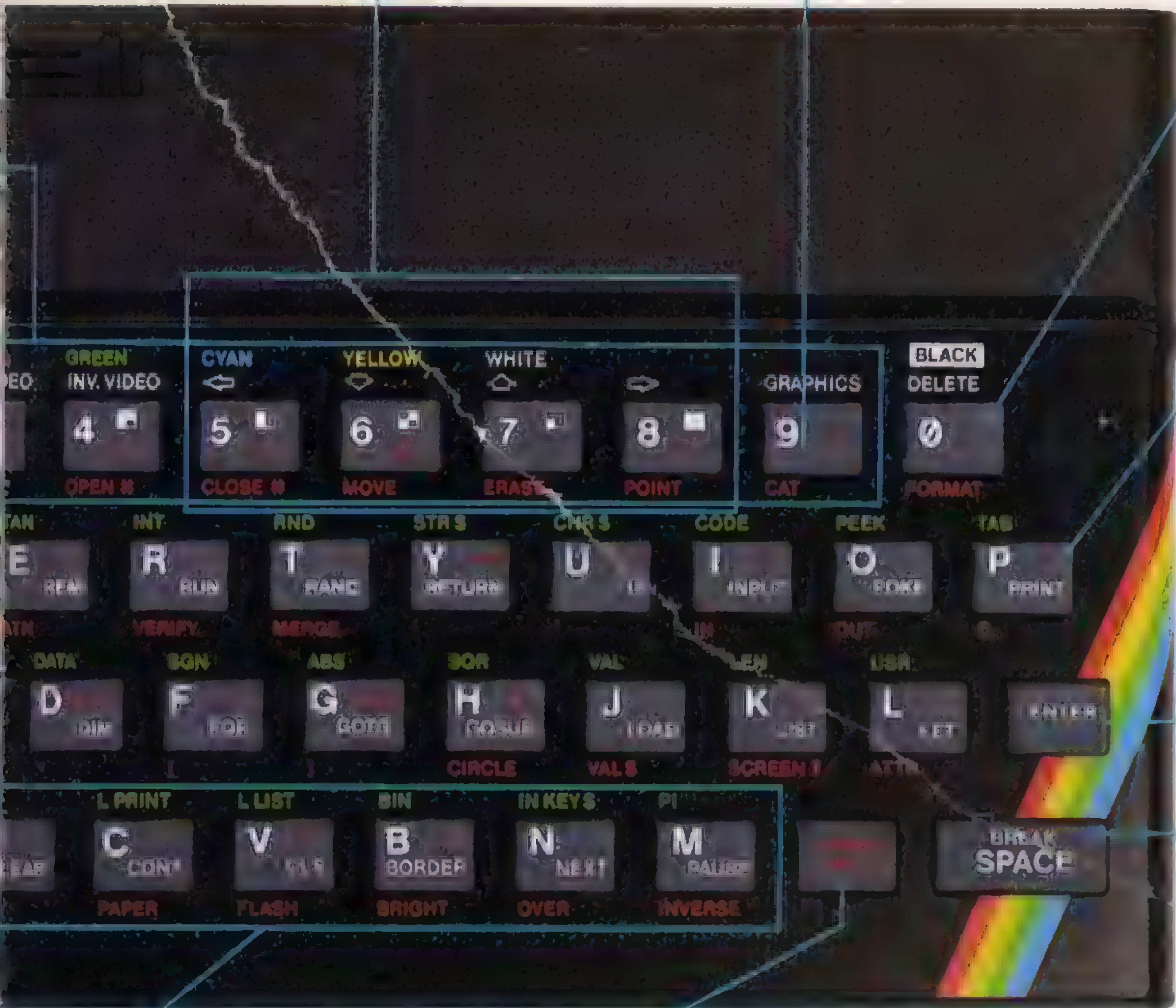
Cursor controls These four keys are used to direct the screen cursor to any point in a program that needs alteration. The cursor function is selected by CAPS SHIFT.

GRAPHICS When used with CAPS SHIFT, this key switches the Spectrum to the graphics mode, ready to accept graphics symbols from keys 1 to 8.

DELETE When used with CAPS SHIFT, this key backspaces the cursor and deletes keywords and symbols on the screen. It also codes for the "colour" black.

PRINT This key carries the frequently-used combination PRINT". Pressing this key once produces PRINT, while pressing it again while pressing SYMBOL SHIFT produces the double quotation marks.

ENTER The Spectrum will not respond to most commands unless they are followed by this key. It is roughly equivalent to the typewriter's carriage return key. When the ENTER key is pressed, the Spectrum will then respond to a command or point out any mistakes in typing.

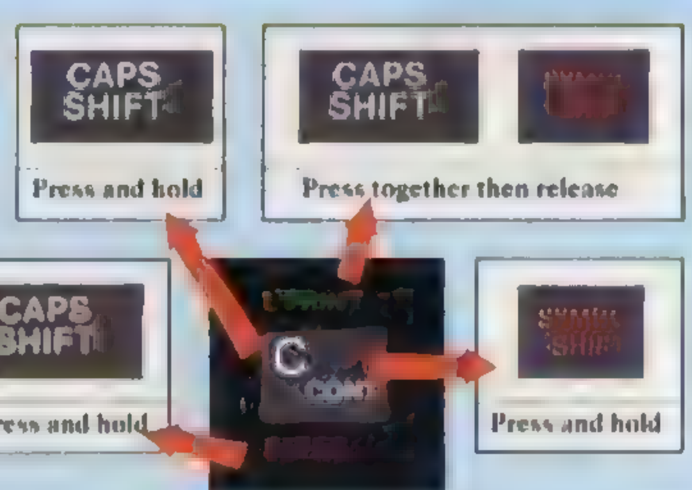


SPACE This is equivalent to the typewriter's space bar. It also has a second function – to BREAK or halt a program before it has finished running. BREAK is selected by CAPS SHIFT.

SYMBOL SHIFT The red symbol on each key is selected by holding this key down while the key required is pressed. It may also be used in combination with CAPS SHIFT.

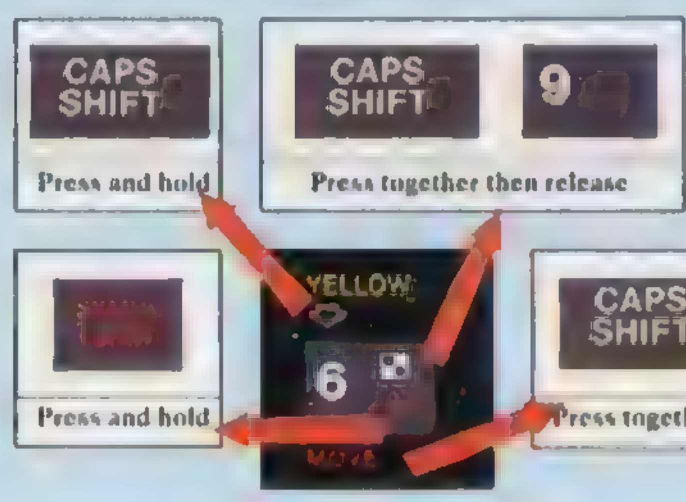
How to select key functions

Letter keys When used with the "K" cursor, a letter key will produce a keyword. The cursor will then change to "L", and the letter key will then produce letters. The keywords above and below the key are produced by combinations of the shift keys.



Unshifted functions If no shift keys are used, a letter key will produce first its white keyword, then the lower case letter.

Number keys When used with CAPS SHIFT and the 9 key, these produce keyboard graphics. The keyword in red below a number key is produced by a combination of shift keys. The red symbol on the key is produced by the SYMBOL SHIFT key.



Unshifted functions If no shift keys are used, a number key will just produce numbers.

SETTING UP

Setting up your Spectrum is quite straightforward and logical. Getting the best possible results on the television screen sometimes takes a little longer. To begin with, connect your Spectrum power supply to the mains and link it to the computer. The computer has no on/off switch. As soon as you connect it to the power supply, you should be able to hear the computer humming if the connection has been made.

Now you need to make a connection between the Spectrum and the television so that you can see the results of your programming. Take the black lead supplied with the computer and plug it into the television's aerial socket. Plug the other end into the Spectrum socket marked "TV", and then switch on the power to the television.

The first results that you see will probably look like a blizzard, accompanied by a loud hissing coming from the television loudspeaker. Turn the television volume control down as far as it will go. The Spectrum will produce all the sound effects you program with its own built-in loudspeaker. Now switch the television to a channel that you can allocate permanently to the computer. The television treats the Spectrum's signal like any ordinary broadcast, so you have to tune the television just as you would to watch a television program. Adjust the tuner controls until you see this on the screen:

SPECTRUM SCREEN SIGNAL

If you cannot get any picture at all from the computer, check that all the power connections have been made properly. Next, check that the Spectrum's TV socket is connected to the television's aerial socket and make sure that the channel you are tuning is the one selected. You should only have to tune your television once. After that just selecting the right channel should produce a correctly tuned display.

How to test the Spectrum's colours

To enable you to test all the Spectrum's colours, you can use this very simple series of commands to see all the colours on screen. Having turned the computer on, press the following sequence of keys (ENTER here indicates the ENTER key at the right of the keyboard):

```
B1 ENTER
B2 ENTER
B3 ENTER
B4 ENTER
B5 ENTER
B6 ENTER
B7 ENTER
```

Every time you press the ENTER key, you should see a change in colour in the "border" area around the screen. (The techniques used to produce colours are explained on pages 34–35). B1 ENTER should colour the screen like this:

B1 COLOUR DISPLAY

When you use this sequence of colour commands to change the colour of the screen, you may find that they seem to have no effect. The tuning required to pick up the Spectrum's colour signal precisely is quite delicate, and you will need to experiment for a while to get the best results. If you do get a picture, but then if you cannot produce colour on the television, there is a possibility that your television is not able to interpret the Spectrum's colour signals. Your dealer should be able to advise you if this is the case.

Although the Spectrum's output is normally viewed on an ordinary television receiver, the signal can be fed into another type of television set known as a monitor. This contains everything that your television receiver has except a tuner, so it cannot receive television broadcasts. Changing the Spectrum's stream of data to

a high-frequency television signal and then reversing the process inside the television reduces the picture quality. By eliminating these stages, a monitor is able to produce better quality pictures. The screen photographs in this book were taken using a monitor, so your own television may produce slightly less clear displays.

Connecting peripherals

The next connection you will want to make is to a tape cassette recorder. The method for using a tape recorder to save your programs is covered in detail on page 60. If you do want to use the cassette recorder, make sure that you have the connecting lead. This is a two-core flex which is coloured black and grey. It is very important that you do not cross-connect this, or the cassette storage and playback will not work.

Later, you may wish to add a printer to your system. The Sinclair ZX printer is supplied with a short cable terminated by a plug which clamps onto the edge connector in the Spectrum's rear panel. It cannot be fitted the wrong way round. Microdrive units are fitted through an interface which sits underneath the computer, again linking up with the edge connector.

When you are using the edge connector, don't force a plug in if you feel any resistance. You may be pushing the plug in wrongly, and this could cause damage.

Arranging the computer To make using the computer as comfortable as possible, the television screen should be lined up behind the computer so that both can be seen without turning your head. The screen should not be too close.

Cassettes used for program storage should be kept away from the power supply, computer and television. Recorded programs may otherwise be disrupted by the magnetic fields produced by these pieces of equipment.

USING THE SCREEN

Having set up your Spectrum, you may already have given in to the temptation to tap a few keys and see what happens. If not, try it – you can't do any damage. The first thing you will notice is that the Sinclair copyright line at the bottom of the screen disappears, and is replaced by a line of characters. But as you will have seen on pages 10–11, exactly which characters appear depends not only on which keys you press, but also on the combination of keys that you use.

Starting to PRINT

To make some sense of this apparently confusing situation, disconnect the power for a second or two to clear the computer's memory and then press the key with PRINT on it (the P key) followed by one of the number keys, and then by the key marked ENTER. As soon as you press the ENTER key, the number you pressed will appear at the top of the screen. You can use PRINT to put a series of numbers on the screen:

PRINT WITH NUMBERS

If, instead of disconnecting the power, you press the key marked CLS (the V key) and then the ENTER key, everything that you have PRINTed will disappear.

What is a variable?

Now try typing in:

PRINT x

followed by the ENTER key. The computer will respond to this – or a command to PRINT any other letter – by displaying an error report:

2 Variable not found, 0:1

This report, one of many that the computer has stored in its permanent memory, indicates why it cannot follow the instruction that you have just given it:

PRINT ERROR REPORT

Instead of putting x on the screen, it has been hunting in vain for something in its memory, a variable.

Using the SYMBOL SHIFT key, now type in:

PRINT "x"

When you press ENTER, the computer makes the correct response – it PRINTs x at the next line.

You have just discovered that to the computer, x on its own and "x" mean two completely different things. The computer treats any letter on its own as a variable. A variable is simply a label identifying a number stored in the computer's memory. To make PRINT x comprehensible to the computer, give x a value (remember to press the ENTER key after each line):

LET x=14
PRINT x

USING **LET** AND **PRINT**

Now x is labelling something, the number 14. LET (on the L key) is a command which gives a label and a value to a slot in the memory. Every time you ask the Spectrum to PRINT x, it will display the last value keyed in. Because x is always a number, it is called a numeric variable.

How to use strings

So x is a numeric variable, but "x" is not; furthermore, even if you substituted a number for "x", it would not become a numeric variable, unless you removed the quotation marks. The computer displays everything inside quotation marks exactly as you type it. You can use any characters on the keyboard – letters, numbers, mathematical symbols and punctuation marks. Type these examples on your keyboard. Remember that you can use PRINT as many times as you like, as long as you press the ENTER key at the end of each command. You can pick capitals with CAPS SHIFT and CAPS LOCK:

```
PRINT "AGE"
PRINT "London"
PRINT "SPECTRUM SCREEN TEST"
```

PRINTING STRINGS

The characters between the quotation marks are called a string. In the same way as a number is stored in the computer and labelled by a numeric variable, a string is stored and labelled by a string variable. String variables are again always letters but unlike numeric variables they always end in a dollar sign. In the line:

```
LET A$="LONDON"
```

A$ is the string variable and LONDON is the string it labels. Having typed in the above line on the keyboard, clear the screen with CLS and then type:

```
PRINT A$
```

After you press the ENTER key, the computer will reveal the contents of the string variable typed in.

Positioning type with TAB and AT

Now try a different sort of PRINT command, this time with a sequence of strings:

```
PRINT "One", "Two", "Three", "Four", "Five",
"Six", "Seven", "Eight"
```

You will find that after you press ENTER the words are PRINTed in two columns. In fact, the screen is divided into two invisible fields, each 16 characters wide. "One" is PRINTed in the first field, "Two" in the next field, "Three" back in the first field, and so on.

However, two further commands, TAB (on the P key) and AT (on the I key), allow you to PRINT characters or strings at any position on the screen. Clear the screen with CLS and then type these two lines (you will need to type out everything between the quotation marks in full using letters):

```
PRINT TAB 10;"TAB EXAMPLE"
PRINT AT 12,10;"x This is AT 12,10"
```

If you press ENTER after each line, your screen should look like this:

USING TAB AND AT

This shows how TAB and AT position the text. TAB is used like the tab setting on a typewriter to PRINT at any position on a line. The number that follows it is the character position. There are 32 positions on a line – the TAB positions for them are numbered from 0 to 31, working from the left.

AT works like TAB, but it allows you to specify a vertical position as well, so you can PRINT at any point on the screen. AT is always used with a pair of numbers separated by a comma. The first number is the line number, working downwards from the top of the screen. There are 22 lines on the screen, numbered from 0 to 21. The second number is the character position, which works just like the number used with TAB. Always remember the comma when typing an AT command, otherwise the computer will not understand it.

COMPUTER CALCULATIONS

The PRINT command is not limited just to displaying characters on the screen. You can also use it in conjunction with the four mathematical functions – addition, subtraction, multiplication and division – to perform calculations that you can follow on your television set.

Let's take addition first. The plus sign is on the K key, the second along from the ENTER key. Because it is the red symbol on the key-top, the SYMBOL SHIFT key must be pressed with the plus key to select the required character. To add two numbers together, simply use PRINT followed by the calculation. Subtraction is carried out in the same way. The minus sign, which doubles as a hyphen when used in text, is on the J key. Like the plus sign, it is also a shifted character. Here are some examples, together with the results they produce. Remember to press the ENTER key at the end of each line to make the computer carry out the calculation:

```
PRINT 6+18
PRINT 250+16.5
PRINT 1.999+6
PRINT 905+139
PRINT 539.7-19.4
PRINT 1842-655
PRINT 4.688-4.666
```

ADDING AND SUBTRACTING

Multiplication is carried out, not with the familiar ×, but with an asterisk, *. The asterisk is the red SYMBOL SHIFT character on the B key on the bottom row of the keyboard. Division uses the oblique stroke, /, a SYMBOL SHIFT character on the V key. In 24/8, for instance, the left-hand number is divided by the right-hand one. To key in the first of the following examples, press PRINT 3*6 and then ENTER:

```
PRINT 3*6
PRINT 14*9
PRINT 2.5*18
PRINT 5*78
PRINT 24/8
PRINT 366/3
PRINT 600/15
PRINT 100/0.01
```

MULTIPLYING AND DIVIDING

Exponents and square roots

In addition to these familiar maths functions, you can raise one number to the power of another, (called exponentiation). The keyboard cannot produce superscripts like the 3 in 2^3, so instead you have to use the up arrow (↑) symbol. 2^3 is calculated by PRINT 2 ↑ 3. Here are some examples of the up arrow in use:

EXPONENTS

The computer also allows you to find out the square root of a number. The command for this is SQR, the green function on the H key. SQR is used like this:

```
PRINT SQR 2
```

When you press ENTER after keying in this line, the computer will PRINT out the answer. However, if you try this command with a minus number, the computer will produce an error report to let you know that you have asked for a mathematical impossibility.

Getting the order right

You can carry out a number of different calculations using the same PRINT command. Try it with addition and subtraction first:

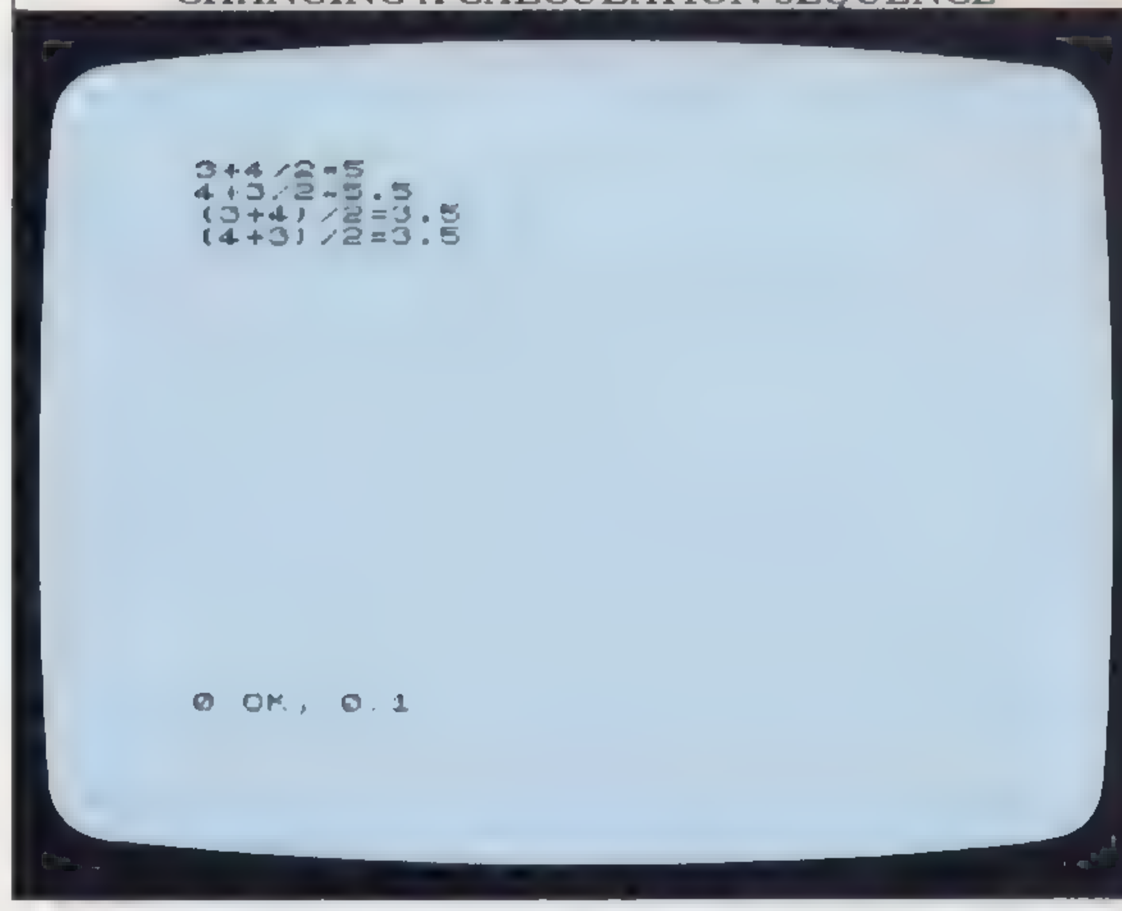
```
PRINT 2+6+3+7−8+3−4
PRINT 48−42+16−2
PRINT 122−19+32+2.5
PRINT 4.8+2.8+1.9
PRINT 1024+14−23
PRINT 15.5−12.5+7.6−3.8
PRINT 56−54+8+34−2+34+92−100+3
```

MULTIPLE CALCULATIONS

You can enter the figures for each calculation in any order at all, and the result will be the same. However, when you add multiplication and division to the chain of calculations, apparently odd things may happen. Look at the next list of examples, and try the calculations for yourself. Say you want to add two numbers together and divide the result by 2. The order in which the numbers are added should not make any difference to the result, but it appears to do so. Each of these lines PRINTs a calculation and result:

```
PRINT "3+4/2=";3+4/2
PRINT "4+3/2=";4+3/2
PRINT "(3+4)/2=";(3+4)/2
PRINT "(4+3)/2=";(4+3)/2
```

CHANGING A CALCULATION SEQUENCE



If 3+4 is exactly the same as 4+3, then why should the subsequent division by 2 make any difference? The reason is that the computer does not necessarily carry out calculations in the order in which you PRINT them on the screen. It performs exponentiation first, then multiplication and division, and finally addition and subtraction, always working from left to right. So in PRINT 3+4/2, 4 is divided by 2 before 3 is added. In PRINT 4+3/2, 3 is divided by 2 before 4 is added.

The problem you set the computer was to add two numbers together and then divide the result by 2. Neither of these examples does that. But you can change the order in which the computer performs calculations by enclosing parts of the calculation inside a pair of round brackets, as in the last two examples on the screen. Here, the addition within the brackets is carried out first and then the result is divided by 2.

Knowing your limitations

There are limits to the numbers that the Spectrum can handle and these limits take two forms – size and accuracy. The size limitation is unlikely to inconvenience you. Positive numbers can have any value from 4×10^{-39} (4 divided by 1 followed by 39 noughts) to about 10^{38} (1 followed by 38 noughts). The Spectrum stores these numbers to an accuracy of 9 or 10 figures. The computer memorizes just the first nine digits – the rest it sets at zero.

You may come across another of the computer's quirks when dealing with very big numbers. The Spectrum does not display them in the way in which you type them on the keyboard. For example, PRINT 2000000000000 produces 2E12 on the screen (the E stands for exponent). This is simply a shorthand way of displaying 2×10^{12} or 2 followed by 12 zeros, the number you keyed in. Try entering PRINT 10, PRINT 100, PRINT 1000 and so on and see how the computer deals with increasingly big numbers.

WRITING YOUR FIRST PROGRAM

So far you have given your Spectrum commands to which it has responded immediately. These commands have been very simple – in many cases it would have been quicker not to have used the computer at all. However, commands on their own are not computer programs. The computer reads each command, carries it out and then forgets it. A program on the other hand is an orderly list of instructions which the computer can store in its memory. It can carry them out as and when you wish, and as many times as you want.

From commands to lines

Having found a task that you want your Spectrum to carry out, the next job is to write the program in steps that the computer can understand. The Spectrum, like most personal computers, uses a computer language called BASIC (Beginners' All-purpose Symbolic Instruction Code). BASIC is an example of a high-level language, a language composed of words and symbols with which you, the user, are already familiar. It is, therefore, an easy programming language to learn.

The commands you key into your computer can be turned into programs by adding line numbers:

USING LINE NUMBERS

As you key the program in, you will notice that the commands are not now carried out as soon as you press the ENTER key, but instead remain displayed on the screen. Now that the program is safely stored in the computer's memory, it only remains to run it and see what it does. Do that by pressing RUN then ENTER.

You may be wondering why the lines are numbered 10, 20 and not 1, 2. When you are writing and testing programs, you will frequently want to add extra lines. If the existing lines are numbered 1, 2, 3, 4, and so on, there is nowhere to put the new lines in sequence. In the above program, there is room to add extra lines numbered 0–9 and 11–19, if necessary.

The program is still in memory, so to try the next one, switch off the power for a second or two to reset the computer and erase the old program. Then see what the following produces when you RUN it:

SCREEN DEMONSTRATION PROGRAM

Taking it from the top, what is a REM? REM is short for REMark. It's a useful device for titling or labelling parts of programs, so that you can find them again quickly. As your programming ability grows, you will find REM lines very useful for reminding you how a particular program works. Other people will also be able to follow your programs more easily if there are periodical REM statements to explain what you are doing. The computer doesn't do anything with a REM statement other than store it in memory.

CLS you have come across already. It's a quick way of taking all the old unwanted information off the screen. Using PRINT on its own (line 30) may at first seem a little crazy. PRINT tells the computer to send whatever follows it to the screen and move on to the beginning of the next line on the screen. If nothing follows PRINT, it just moves to the next line. Here, it is used to move the line of hypens down.

How to correct mistakes

It is easy to make mistakes on the keyboard which prevent the program from working. The Spectrum will not allow you to ENTER a line that contains a mistake in BASIC, but this doesn't stop you writing a program with correct lines that produces the wrong result. If you do notice a line that needs changing, there is no need to retype the program. The computer always uses the last version of any line, so to make a change, simply retype the line at the end of the program, press ENTER and the computer will insert it in the correct place.

Why punctuation is important

The next program uses the techniques demonstrated on pages 16–17. Switch off the power again for a second or two before keying it in:

CALCULATIONS PROGRAM

```
10 PRINT "3*18=";3*18
20 PRINT "6.25*4=";6.25*4
30>PRINT "179*9.8=";179*9.8
```

In each of the three calculations, the screen first shows what the calculation is (everything within the quotation marks is displayed exactly as it is written), and then the computer carries out the calculation itself and shows the result on the same line. The semi-colon is very important. It ensures that the result is shown on the same line as, and immediately following, the details of the calculation. Try the same program with a comma, a colon and nothing at all instead of the semi-colon. You'll quickly realize how important punctuation is in computer programming. Correct spacing is also vital if you want to produce a readable display when the computer PRINTs numbers and strings following each other. The following program, which again combines some calculations with PRINT, shows spaces within strings and how they appear when the program is RUN:

CONVERSIONS PROGRAM

```
10 PRINT "CONVERSIONS"
20 PRINT
30 PRINT
40 PRINT "1 foot=";12*2.54;" centimetres"
50 PRINT
60 PRINT "1 gallon=";8/1.76;" litres"
70 PRINT
80 PRINT "10 kilometres=";10*5/8;" miles"
90 PRINT
100 PRINT "1 day=";24*60*60;" seconds"
```

CONVERSIONS DISPLAY

```
CONVERSIONS

1 foot=30.48 centimetres
1 gallon=4.5454545 litres
10 kilometres=6.25 miles
1 day=86400 seconds



0 OK, 100 1
```

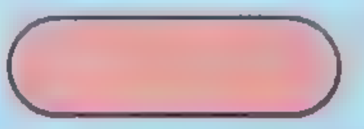
How to write a flowchart

If a program is to RUN properly, it must carry out the correct operations in the right order. Drawing a flowchart is a useful way of outlining the steps involved in making the computer perform a task. This flowchart shows how to plan a program to add up all the numbers from 1 to 1000. Each shape is a separate operation, and the arrows connecting the shapes show the path that the program is to follow. NUMBER and TOTAL represent figures that can be entered in a program as numeric variables – n and t. This program contains two features which you will encounter later – a program "loop" and a program "decision point".

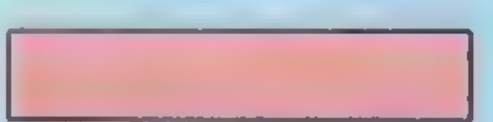
DRAWING A FLOWCHART

This chart shows all the steps needed to program a computer to add together all the numbers from 1 to 1000.

Key



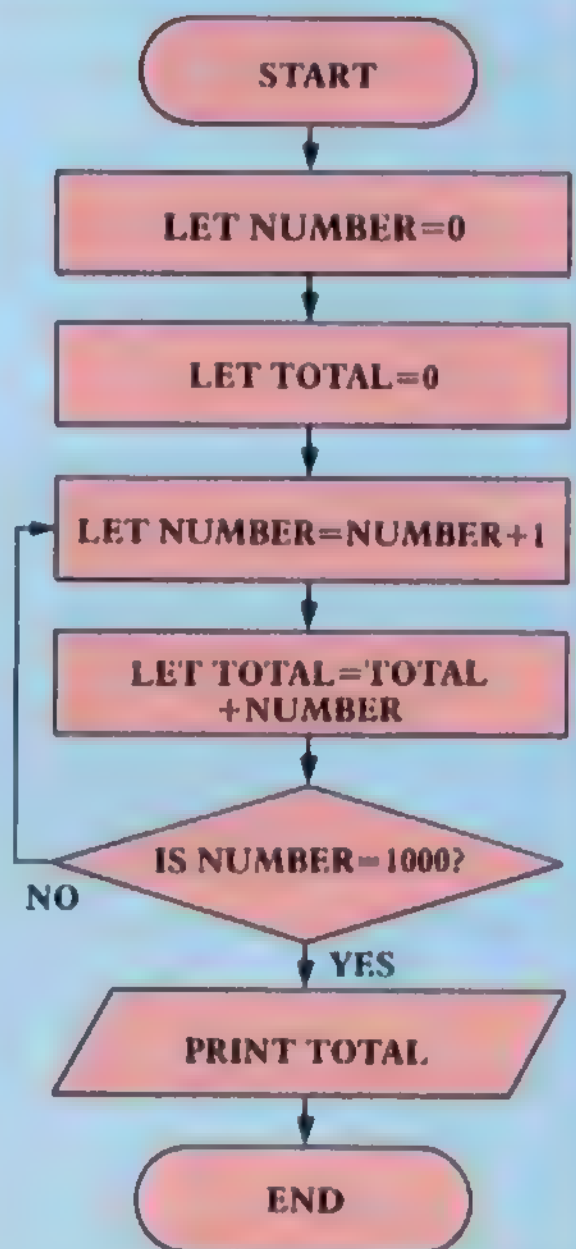
Terminator Signals beginning and end of flowchart



Instruction Identifies each separate operation

Decision point Instructs computer to make a decision

Input/Output Instructs computer to take in or give out information



DISPLAYING YOUR PROGRAMS

As you start writing programs, you will often want to refer back to check on something or perhaps alter it in some way. In order to do that, you must be able to see the program on the screen again after it has been RUN. The Spectrum allows you look at anything you have stored in its memory. In this case, you want to look at the "program listing" – the program as you typed it in.

If you've just switched the computer on again after a short break, type in a program from a previous page. The BASIC word (or keyword) LIST will call up your program onto the screen again from the part of the memory where it is currently stored. Every time you press LIST the program will be displayed once more:

LISTING A PROGRAM

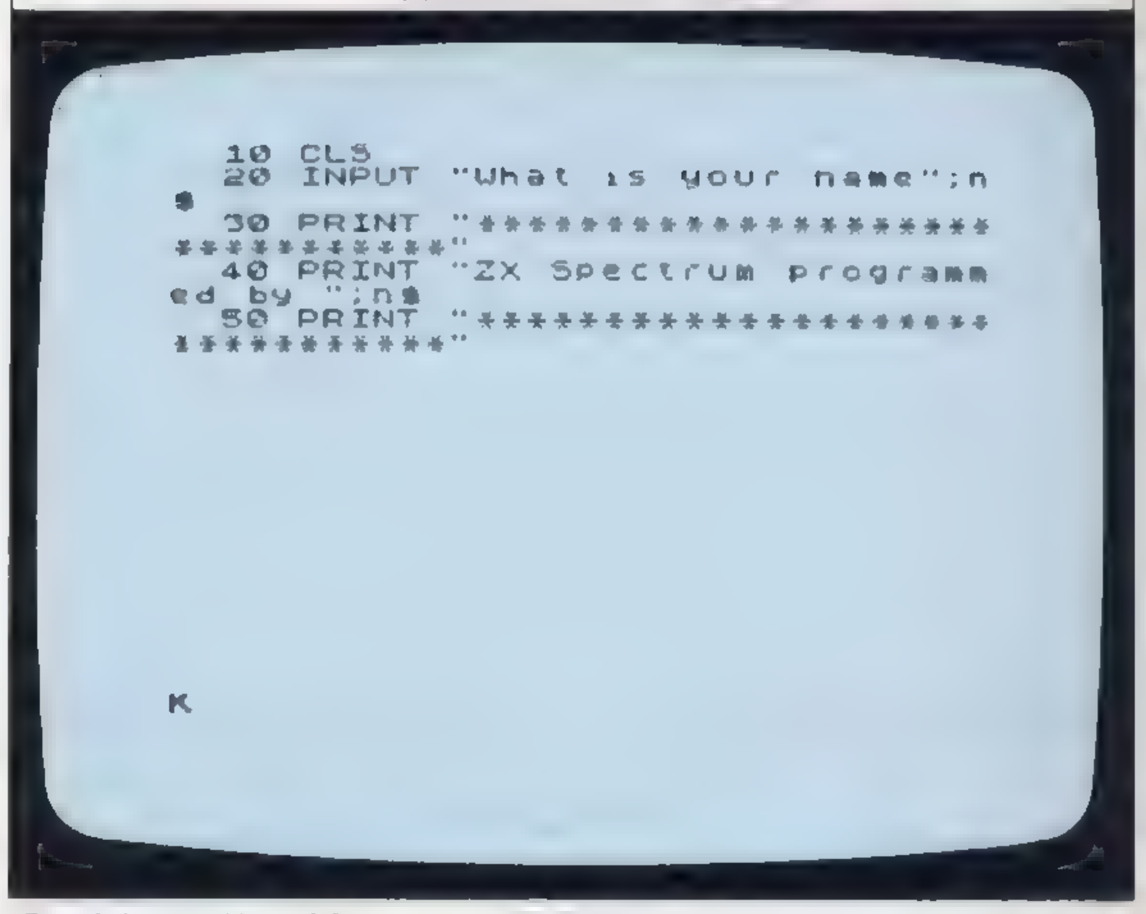
You haven't transferred the program from one part of the memory to the screen – it's still held in memory. You are now looking at a copy of it. If you want to make sure of that, use CLS again to clear the screen. If you now press LIST, the program will then reappear. You can do this as often as you like and the program will remain safely in memory unless you disconnect the computer.

Moving around a LIST

You will notice that when a program has been LISTed, it is displayed in an indented form, with the lines beginning two characters further to the right than they would if you had just keyed them in. Most of the programs in this book are shown in their LISTed form.

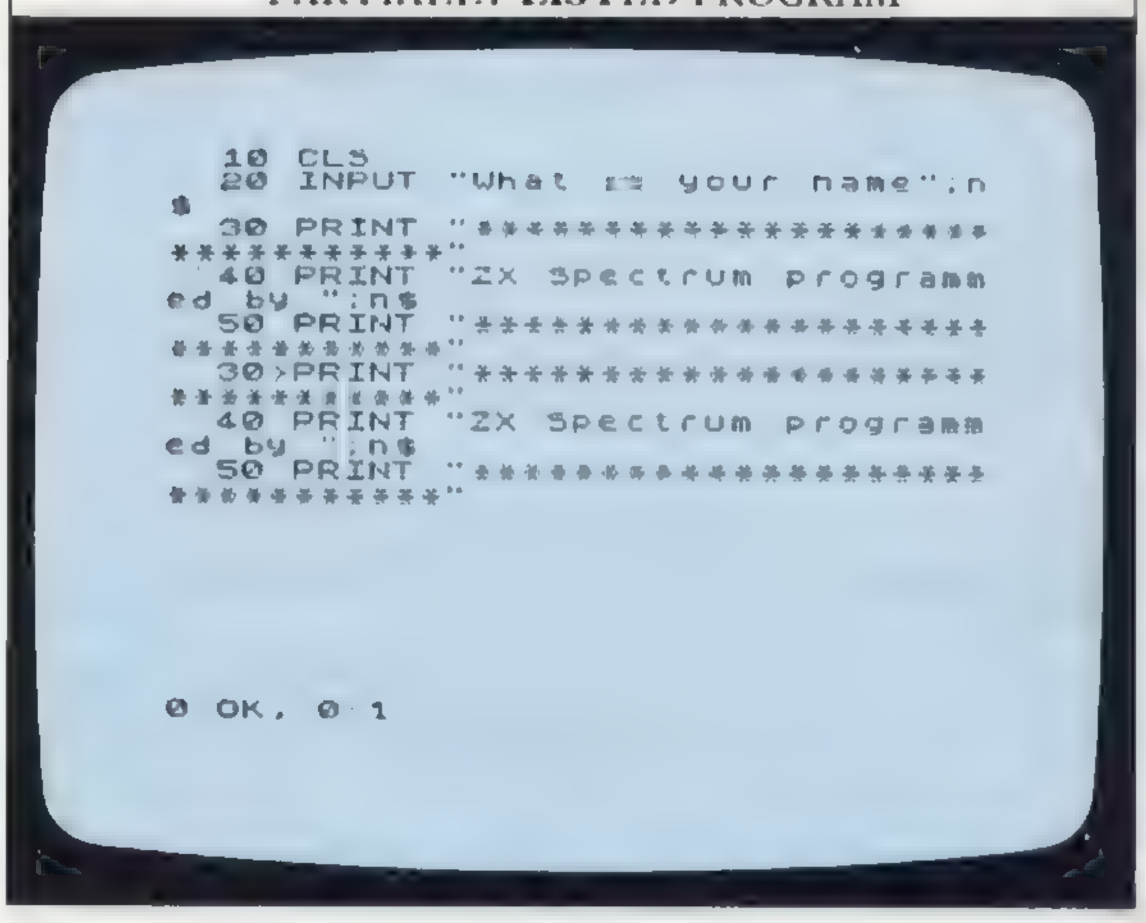
LIST is very useful for developing a program. All you have to do to see your program after RUNning is to type in LIST followed by the ENTER key. But LIST's capabilities do not end with producing a whole program for you to examine. If you key in the next program, you will be able to see how to use LIST more selectively. The program also demonstrates a technique which you will soon be using:

OPERATOR PROGRAM



(Incidentally, if you do RUN this program, press the ENTER key after you type in your name). Now if you want to display the whole program listing, type LIST. But you might only need to see a few lines from the end of a long program. Using the above program as an example, press LIST 30. Only lines 30 to the end of the program appear the second time:

PARTIALLY LISTED PROGRAM



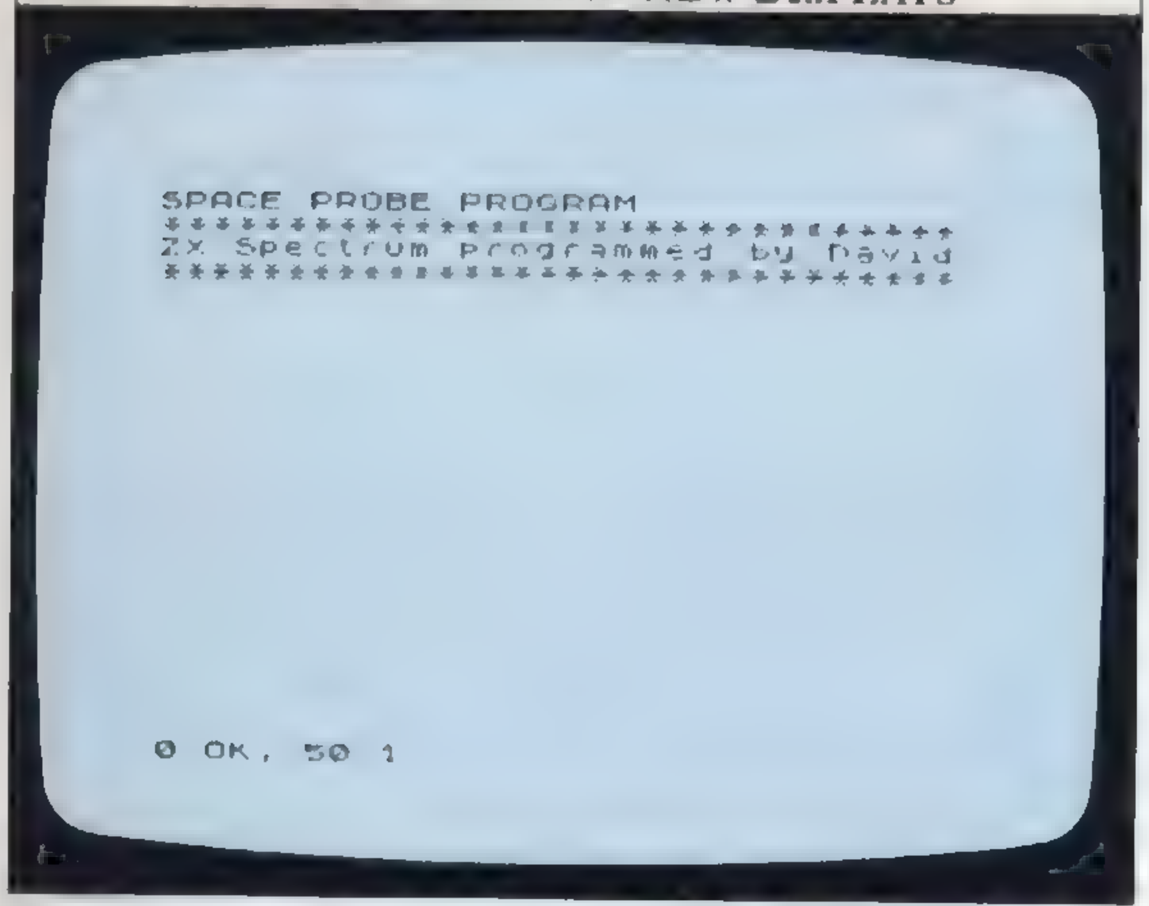
How to enter a new program

Imagine you are starting a new program. Clear the screen and type in the first line:

```
10 PRINT "SPACE PROBE PROGRAM"
```

and RUN it. Something odd happens. The old program is still in memory. The computer PRINTs your new line 10, but then goes on to RUN the remainder of old program, because you haven't erased it:

COMBINED OLD AND NEW DISPLAYS

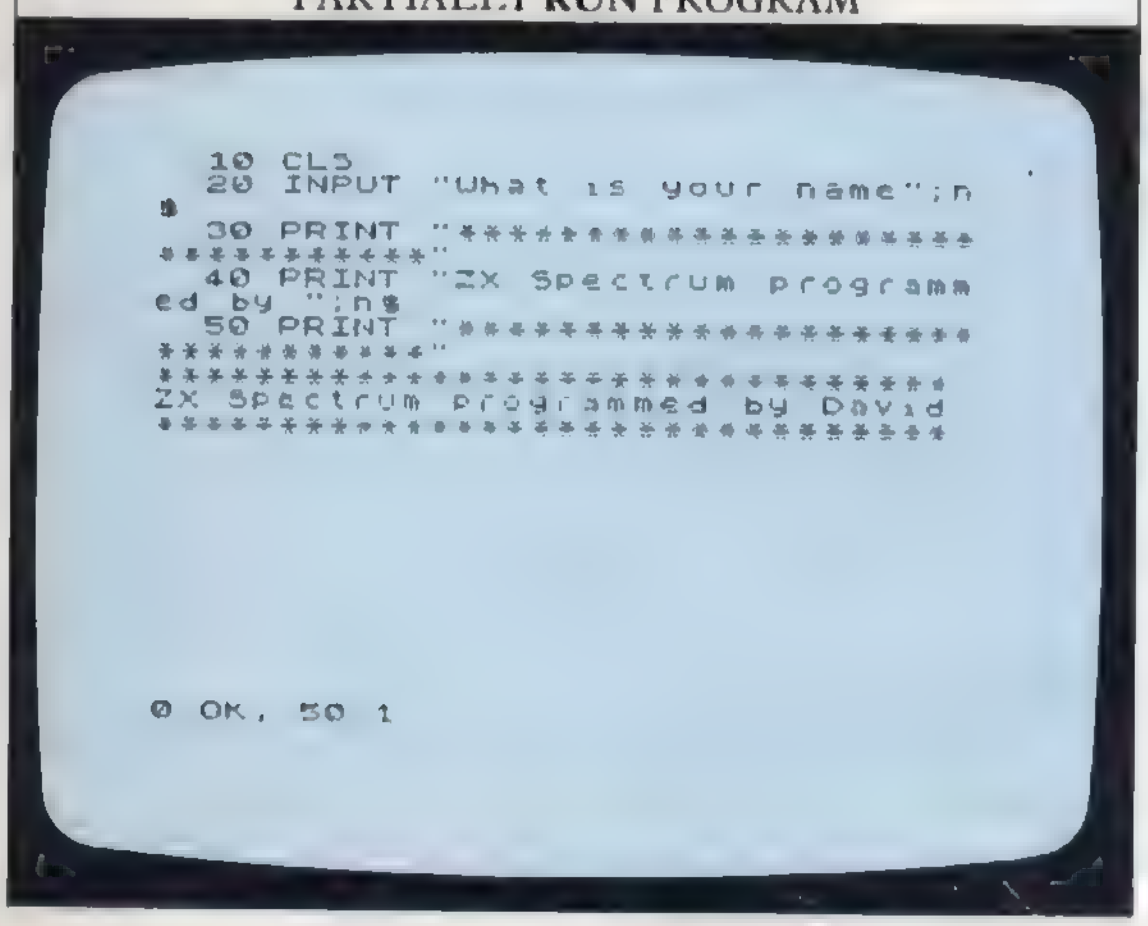


Up to now you have been switching off the power for a second or two before entering a new program, but there are better ways of getting rid of old programs. One of them is to use the BASIC keyword NEW. Press NEW and the ENTER key, then key in the new line 10. This time the old program will have gone for good.

RUNning a program segment

Programs may be RUN starting at any line number you like. Often you may have trouble getting part of a program to RUN properly. In a short program, it's just as convenient to RUN the whole program as it is to RUN only a part of it. But what if the troublesome part that you want to experiment with and RUN over and over again comes near the end of a long program? It soon becomes tiresome and time-wasting to have to watch the first five minutes or so of the program unfold before the suspect part comes up. Fortunately, you can jump part of a program by using RUN and a line number:

PARTIALLY RUN PROGRAM



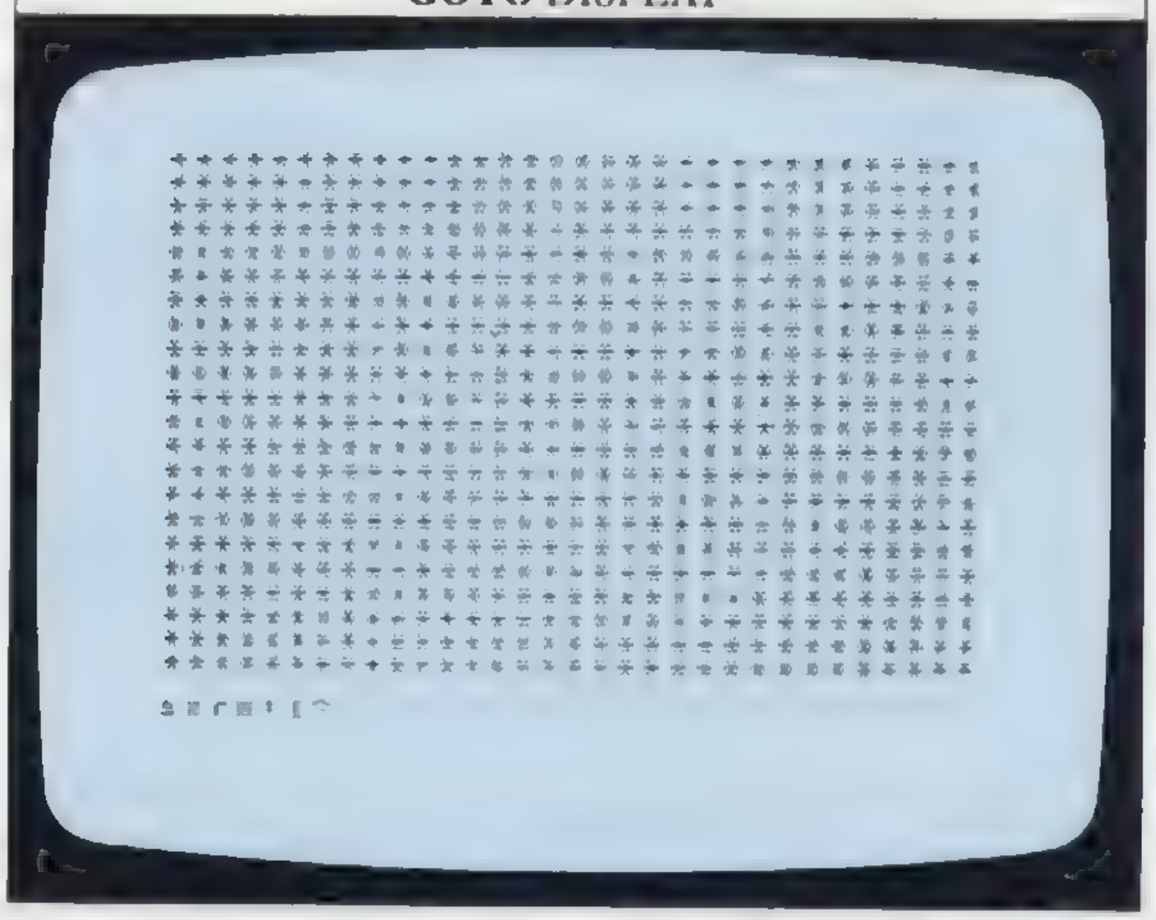
In this screen, the "operator" program from the opposite page has been LISTed and then followed by RUN 30 (produced by pressing RUN and 30 then the ENTER key). The result is that the computer goes straight to line 30, and then continues through the program as before. If you have already RUN the program, n$ will have a value – your name – and the computer will PRINT this out, although lines 10 and 20 will be ignored.

You can get exactly the same effect with the keyword GOTO. GOTO is one of the simplest and most useful commands in the BASIC language. Used without a program line in front of it, GOTO makes the computer go straight to a specified line and then RUN a program from that point. But when GOTO is actually part of a program, the results become very interesting. You can get an idea of what this command can do simply by keying in this simple program:

```
10 PRINT "*";
20 GOTO 10
```

Your screen should fill up with a display like this:

GOTO DISPLAY



If you are puzzled about why this has happened, don't worry. We will be returning to GOTO soon after you have mastered a few more BASIC keywords. However, you will find that it seems difficult to get rid of this display. The clue here is the word "scroll?" that appears at the bottom of the screen. The computer is asking you if you want the characters on the screen to continue to "scroll" upwards to fill the screen again because the program has not finished RUNning. In fact it has no end at all. To clear the screen press the N key. A message will then appear:

```
D BREAK – CONT repeats 10:1
```

Pressing any other key just makes the scrolling continue. Once this message has appeared, you can then LIST or erase the program with NEW.

CORRECTING MISTAKES

Computer programming is one pastime in which mistakes are unavoidable. Programs very rarely work satisfactorily first time, and the longer they are, the more difficult it is to get them right. It's important to realize that making mistakes and correcting them is often an interesting part of program development. So, don't ignore, hide or gloss over your mistakes – they are an invaluable aid to learning how to get things right.

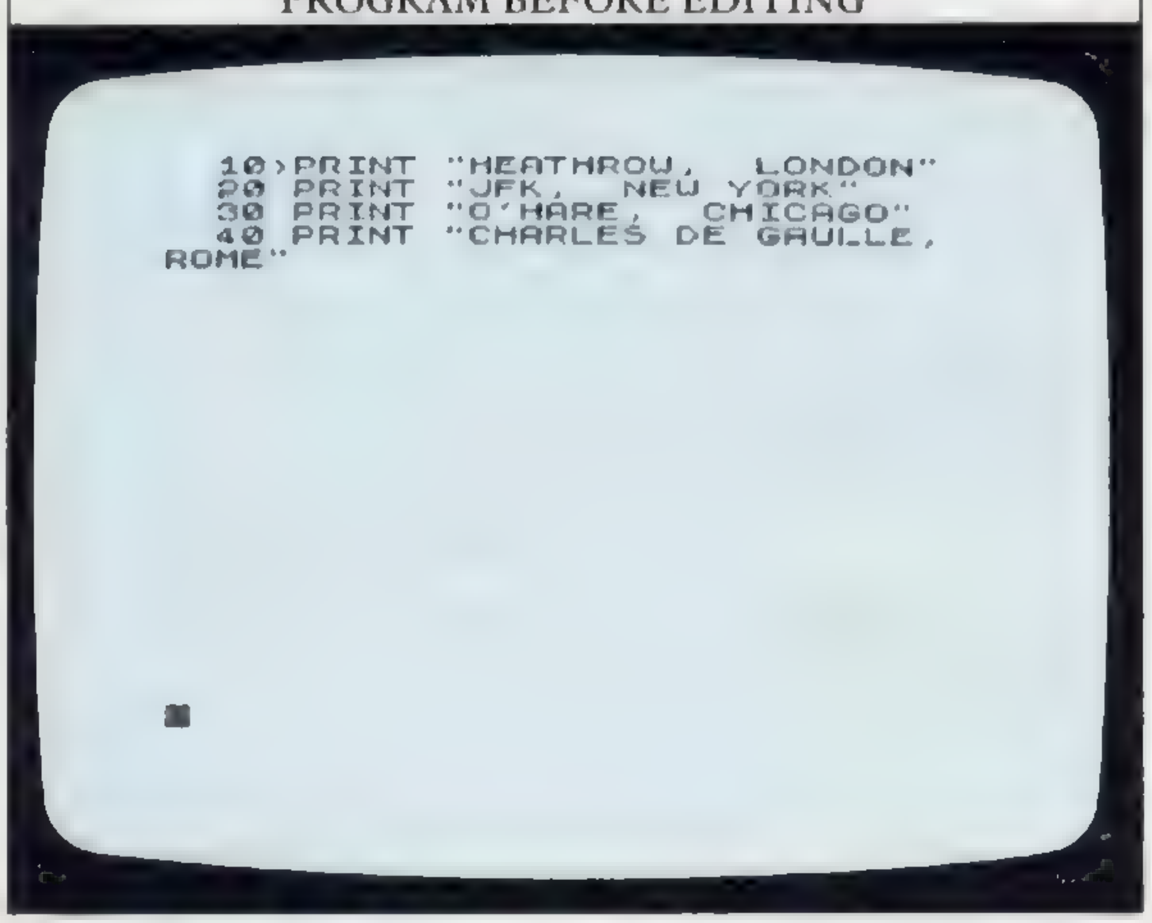
For instance, in a computer program you cannot alter punctuation without completely changing the sense of what you have written. As you saw on page 19, it will have drastic results. To the computer, punctuation means something very precise, and if you get it wrong, a program may not work.

You can change a line in a program in two ways. First, as you have seen, you can simply retype the line. The new version automatically replaces the old one in the computer's memory. However, if there is very little wrong with a line, especially if it is a long line, it's a waste of time to completely retype it. The alternative way of making a change in this case is to use the cursor keys to edit on screen.

Editing on the screen

Editing involves using the four keys with arrows printed above them and the key with EDIT printed above it, all on the top row of the keyboard. Here is a program that needs editing:

PROGRAM BEFORE EDITING



To correct line 40 in the program to read:

```
40 PRINT "CHARLES DE GAULLE, PARIS"
```

you could retype the line. But try using the screen editor instead. First, type in the program and RUN it. Now LIST the program on the screen. Press the CAPS SHIFT key together with the key on the top row of the keyboard with a downward-pointing arrow (not the exponent key featured on page 16). The > symbol at the beginning of the last program line moves to line 10. Use the downward arrow to move it to the incorrect line. Now press CAPS SHIFT and EDIT (top row). The line marked by the > appears at the bottom of the screen:

PROGRAM DURING EDITING

You can now use the left- and right-pointing arrow keys on the top row to move the cursor to the end of the section to be deleted, after the word ROME. Now press CAPS SHIFT and DELETE (top row) repeatedly until ROME disappears. Then simply type in the characters that are to replace ROME. Finally press ENTER. It sounds complicated, but once you've tried it, it soon becomes second nature.

You will frequently want to add lines to a program after you have written the first draft. Perhaps you forgot to put CLS at the beginning to start the program off on a clear screen. You do not have to edit any line numbers to do this. In the above program, for example, you can enter the new first line by typing:

```
5 CLS
```

As the computer executes BASIC instructions in line number order, it doesn't matter that this line was added last – it will be carried out first.

First steps in bug-hunting

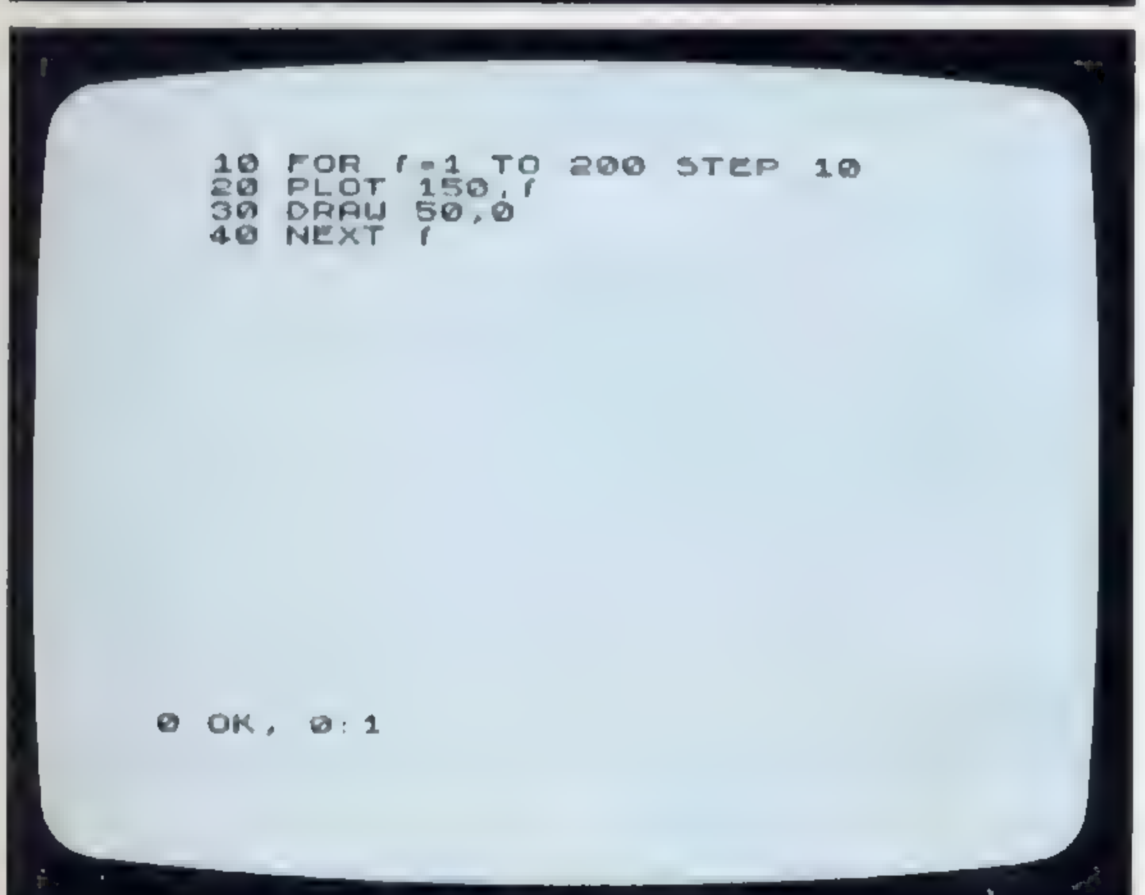
Mistakes in programs are called bugs, and the business of getting rid of them, debugging. As you have probably discovered, the Spectrum helps a great deal in debugging programs by examining what you type in for errors in spelling and grammar or syntax. If it finds any, it alerts you in two ways. First, if it comes across something that doesn't make sense in a line as you are typing it in, it will not let you ENTER the line. Pressing

ENTER will have no effect, and second, a question mark will flash next to the suspect part of the line, giving you a chance to correct it.

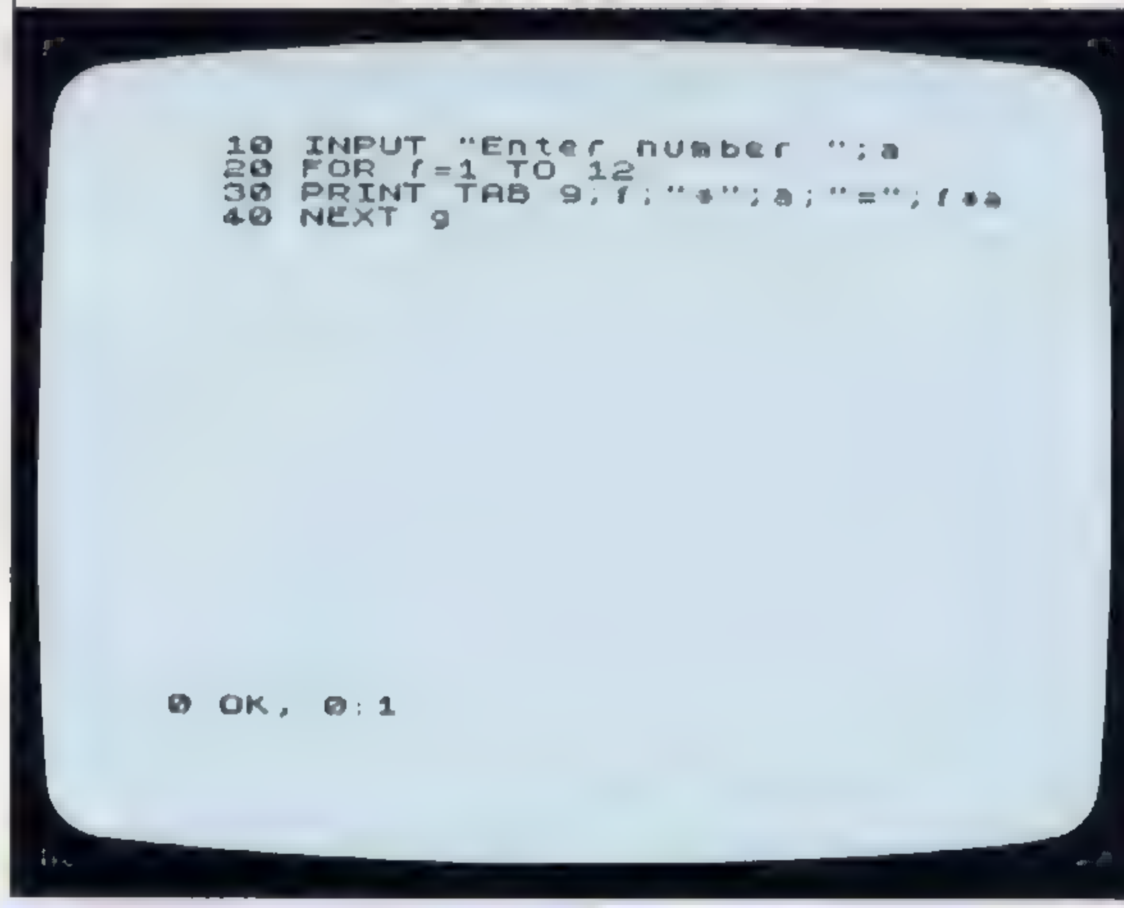
However, even if every single line of your program makes sense to the computer, the program may not RUN properly. You may have inadvertently told the computer to do something impossible – to add two numbers, A and B, when you haven't given A or B values, for instance. It responds to this by displaying an error report on the screen. You came across one on page 14 – "2 Variable not found, 0:1". In fact, the Spectrum can display over 25 different reports.

Each report begins with a number or letter (0–9 or A–R). The report itself is followed by something like 30:1. This means that the program has stopped at line 30. The "1" indicates that the error is in the first statement on the line. (As you'll see later, it is possible to write more than one instruction on a single program line). Here are some slightly more advanced programs which will not work. Try checking the error reports they produce on the table that follows them:

"BUGGED" PROGRAMS



"BUGGED" PROGRAM



ERROR TABLE

These are some error reports that you may encounter when writing your first programs.

Code	Report	Occurrence
0	OK	The "correct" report. The computer has found no errors in your program.
2	Variable not found	You have programmed the computer to do something with a variable without first defining it. This may occur if you miss out quotation marks before and after a string (see page 14).
4	Out of memory	All the available memory has been used up. With the 48K Spectrum this is unlikely to happen unless you try to combine a number of long programs with consecutive line numbers.
5	Out of screen	Your program has tried to INPUT more than a screenful of lines, or has tried to PRINT below the bottom line (see pages 15, 24–5).
6	Number too big	A calculation has produced a number bigger than the computer's limit of 10^{38} (see page 17).
A	Invalid argument	Your program has followed a function with an "argument" (the number on which the function is to operate) which cannot be used – for example SQR followed by a minus number (see page 17).
B	Integer out of range	Your program has produced a number that is larger than the limit that the computer can use for a certain operation. This often occurs with graphics (see page 28).
D	BREAK-CONT repeats	Appears when N, SPACE or STOP is pressed in response to "scroll?" (see page 21). This report also appears if BREAK is pressed while the computer is doing something other than carrying out a program – for example, LOADing a cassette (see page 60).
G	No room for line	The computer's memory has been filled, and there is no room for the line that you have just tried to ENTER.
L	BREAK into program	Appears when the BREAK key is pressed while a program is RUNning. The line and statement numbers which follow the report show the last line to be carried out. Pressing CONTINUE will make the computer carry on from that point.

COMPUTER CONVERSATIONS

In all the programs you have written so far, you have given the computer a set of instructions and left it to carry them out. Each program has had just one outcome, which was exactly the same every time the program was RUN. But few real programs are like this; in a games program for example, the players feed the computer with new instructions every time the game RUNs. The computer takes in these instructions during the course of the game, changing the display in response to the input of information.

Indeed, it is difficult to write a program of any complexity without being able to interrupt the program while it is RUNning to feed in new information.

The BASIC word INPUT is intended to deal with this situation. It lets you carry on a conversation of sorts with the computer – you "talk" to it through the keyboard and it "talks" to you through the screen.

The INPUT command makes the computer remember information typed in on the keyboard, and gives it a name – a numeric variable if the information is a number, or a string variable if the information is in string form. The information is then used later in a program. Here is an example of INPUT at work:

USING **INPUT**

```
  10 CLS
  20 PRINT "What is your name?"
  30 INPUT n$
  40 PRINT "********************
************"
  50 PRINT "ZX Spectrum programm
ed by ";n$
  60 PRINT "********************
************"



0 OK, 0 1
```

Questions from your computer

The program instructs the computer to display the question "What is your name?". Line 30 then stops the program, leaving the question PRINTed on the screen. The computer is waiting for new information from you. There's no need to hurry – there isn't a time limit. The computer will wait forever or until you type in the information it needs. Type in your name and press the ENTER key. The program continues.

The INPUT line of the program takes your name and labels it with the string variable n$. The dollar sign shows that the computer has been programmed to expect a string. This program is similar to the one used on page 20 as an example of LIST. You can see from that earlier program that INPUT can also PRINT:

COMBINING **INPUT** AND **PRINT**

```
  10 CLS
  20 INPUT "What is your name";n
$
  30 PRINT "********************
************"
  40 PRINT "ZX Spectrum programm
ed by ";n$
  50 PRINT "********************
************"



K
```

Programming multiple INPUTs

Many programs use INPUT a number of times to gather different items of information. It is quite easy to do this. All you have to remember is that you will need a separate variable for each INPUT. Once you have given the computer the information that each variable will label, it can then use the variables in a program.

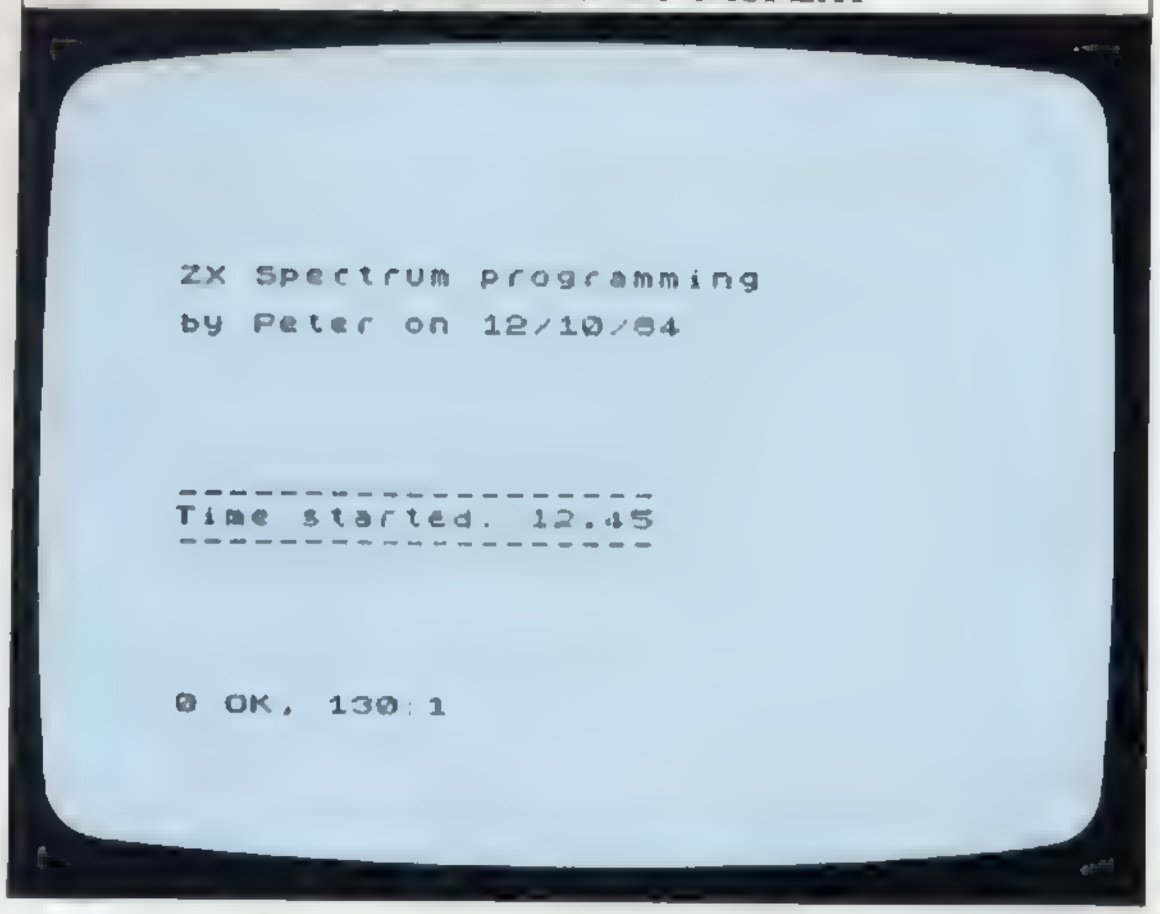
In the previous program, n$ was used to label a string – in that case it was a name. But a string doesn't have to be just letters, it can be numbers. If you label a number as a string, the computer will deal with it as a string. Here is a program that does this:

MULTIPLE **INPUT** PROGRAM

```
  10 CLS
  20 PRINT AT 2,9;"Enter name"
  30 INPUT n$
  40 PRINT AT 5,1;"Enter today's
 date 00/00/00"
  50 INPUT d$
  60 PRINT AT 8,6;"Enter time 00
.00"
  70 INPUT t$
  80 CLS
  90 PRINT AT 5,0;"ZX Spectrum p
rogramming"
 100 PRINT AT 7,0;"by ";n$;" on
";d$
 110 PRINT AT 14,0;"-------------
-------"
 120 PRINT AT 15,0;"Time started
: ";t$
 130 PRINT AT 16,0;"-------------
-------"



0 OK, 0 1
```

In line 100, the computer PRINTs d$, which is the date. If the variable had been just d, the computer would have taken the date's oblique lines to mean "divide by". As a string, d$ is left unaltered:

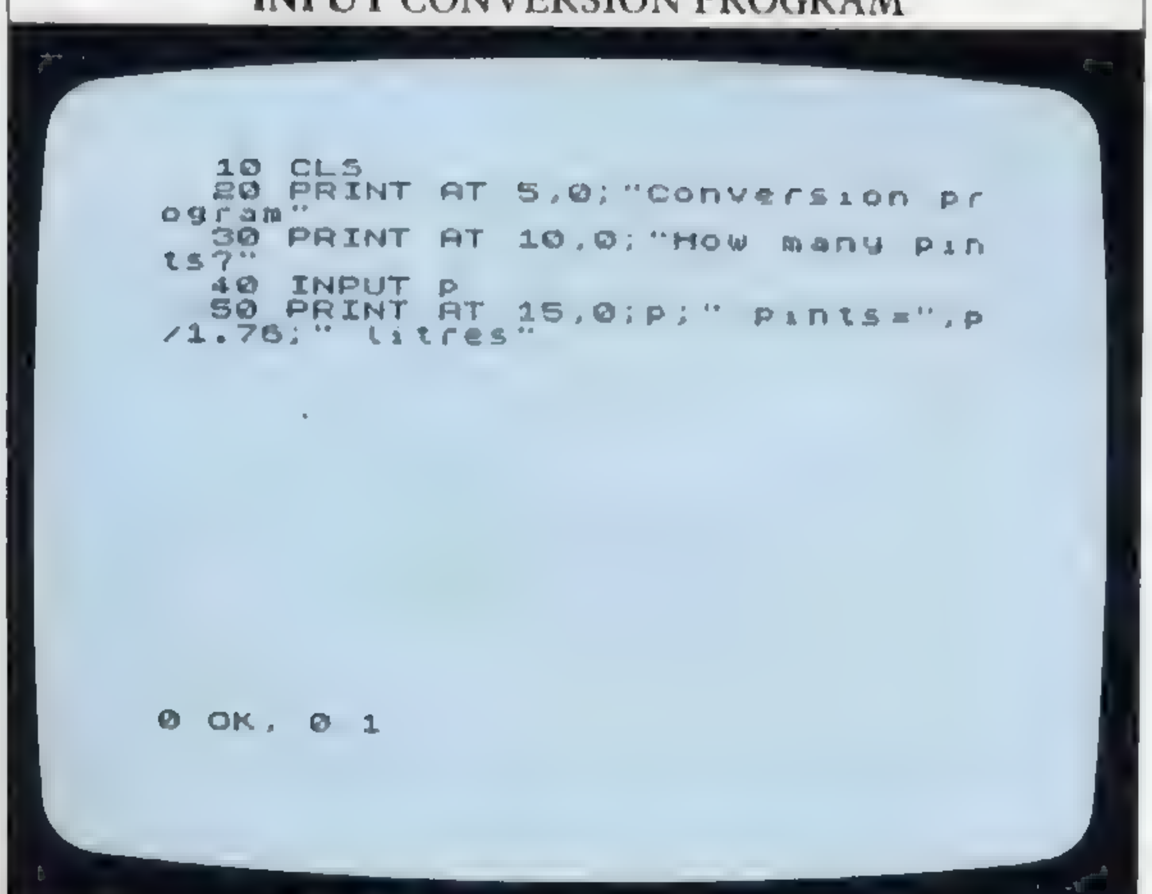
MULTIPLE **INPUT** DISPLAY



Using INPUT with numbers

Because you can use INPUT to gather numbers as a program is RUN, this command has many practical applications. Consider, for example, the problem of converting lengths, sizes or weights from one unit of measurement into another. The conversion is always the same – 2.54 centimetres to the inch, 2.2 lbs to the kilogram, 1.76 pints to the litre, and so on – but the numbers in each new calculation are different. Here is a simple conversion program for you to try out:

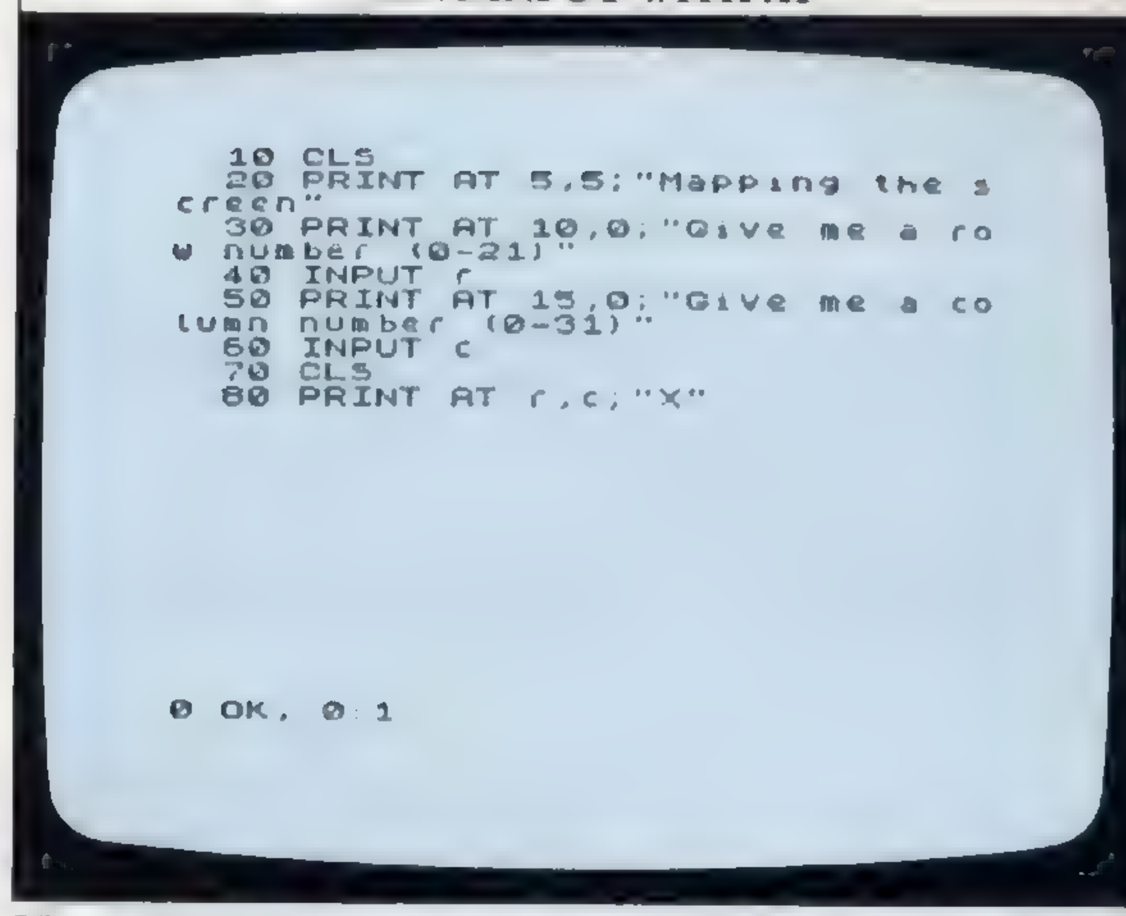
INPUT CONVERSION PROGRAM



The program asks you how many pints you want to convert to litres, waits for your response, does the calculation and then displays the result on the screen. Because the INPUT line is expecting a number in response to the question it asks, the variable it produces is a numeric one, p. This labels the number you key in for use later on in the program.

Next is a program that asks you for two pieces of information, and then uses them with the command AT to fix a point on the screen:

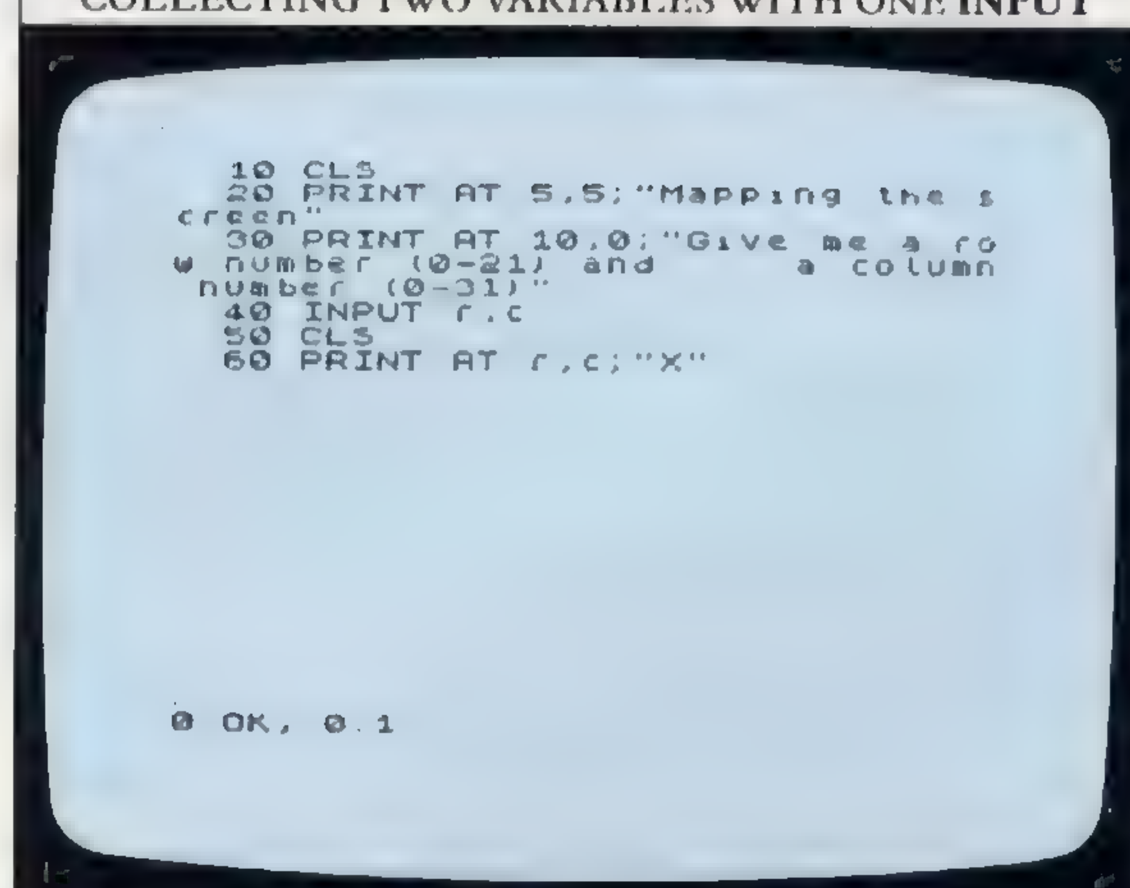
USING **INPUT WITH AT**



If you RUN this program, you will find that it asks you for a row and column number and then PRINTs an X at the position specified by your co-ordinates. The letters r and c are just labels, numeric variables waiting to be given values. It is these values that you key in when the program is RUN.

You can shorten this program by making a single line with one INPUT statement collect both these figures. Type in each number followed by ENTER:

COLLECTING TWO VARIABLES WITH ONE **INPUT**

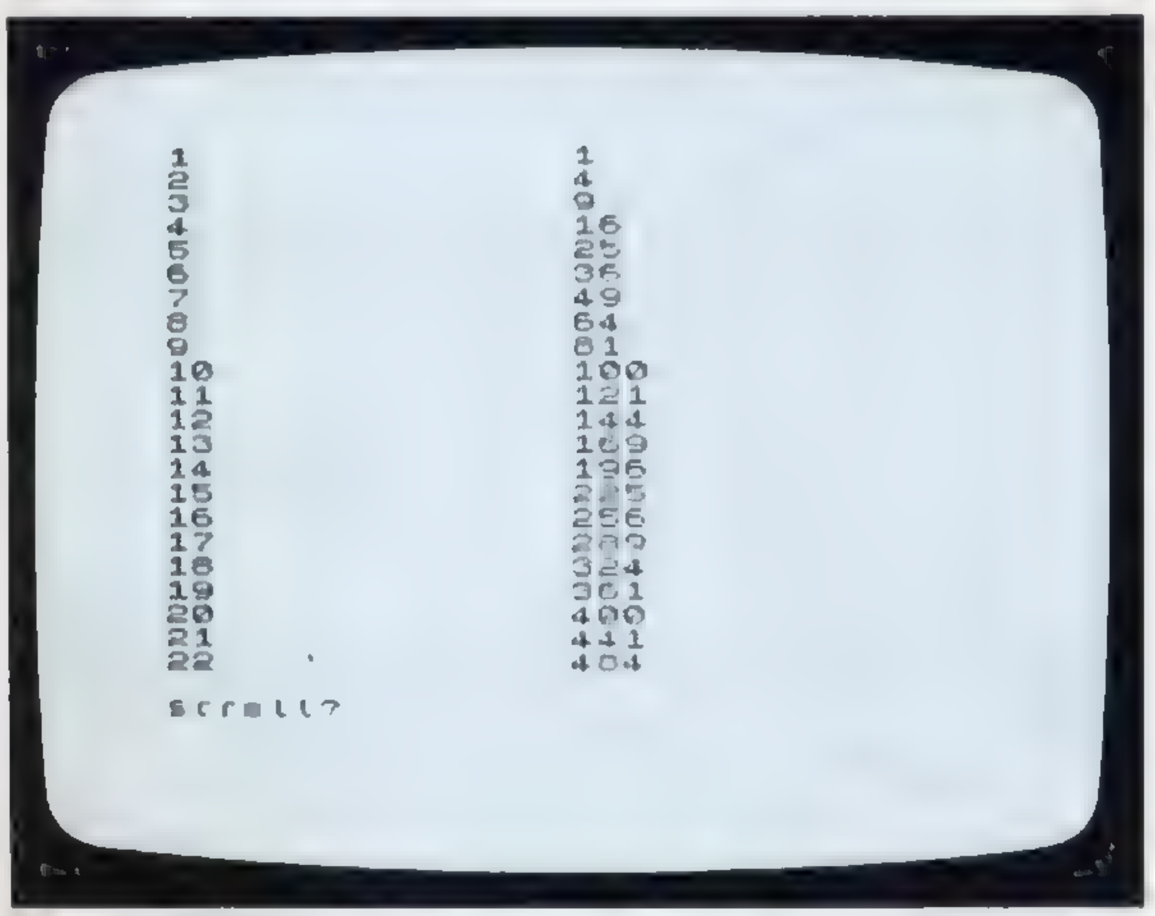


All the spaces between "and" and "a column" may look rather strange to you. If you don't put them in, you'll find that the message PRINTed by line 30 is split awkwardly between two lines. The extra spaces bring the whole of "a column number (0–31)" down to the middle of the next line.

WRITING PROGRAM LOOPS

Computers are extremely good at doing lots of simple, repetitive jobs very quickly. But if it is to do anything involving repetition, the computer must have some means of carrying out the same program or part of a program repeatedly. There are several ways of writing these program "loops". On page 21 you came across a loop using GOTO. Here it is in a slightly more complex loop produced by the same method:

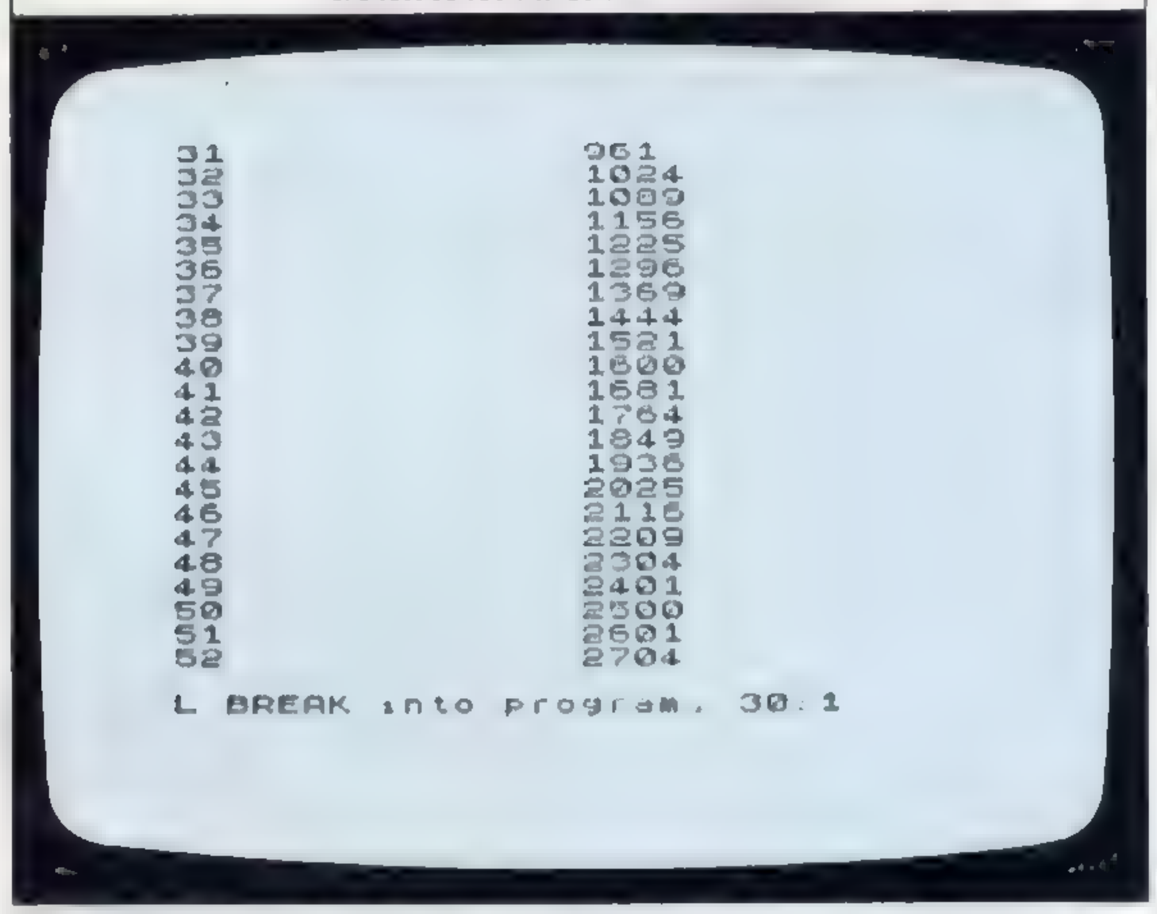
NEVER-ENDING LOOP PROGRAM



If you RUN this program, you will quickly see the disadvantage of using GOTO alone – the program is never-ending. When the screen has filled up with figures, "scroll?" appears at the bottom to show you that the program has not finished yet.

If you press most keys, the display will continue. But if you do let the loop continue scrolling, there is a way that you can later exit from the program. As you saw on page 21, pressing the N key will break the loop. Using BREAK will also do this, producing a line number:

BREAKING INTO A LOOP

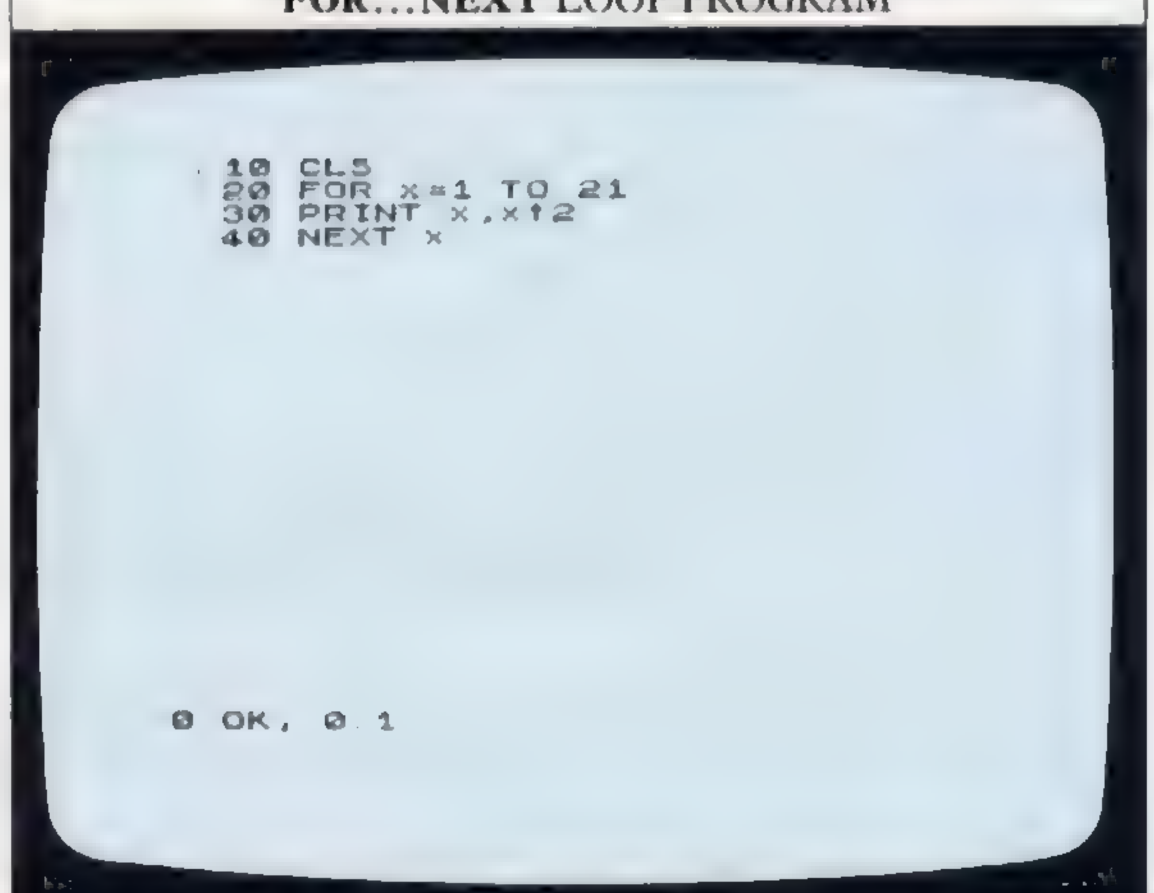


Note that in this loop program, line 50 does not point back to the very beginning of the program at line 10. If it did, x would always be equal to 1, and the screen would clear each time the first line was PRINTed.

How to stop a loop

The solution to these endless programs is the FOR … NEXT loop. This allows you to set limits on how many times the program is carried out. You can adapt the GOTO program to use FOR…NEXT instead:

FOR…NEXT LOOP PROGRAM

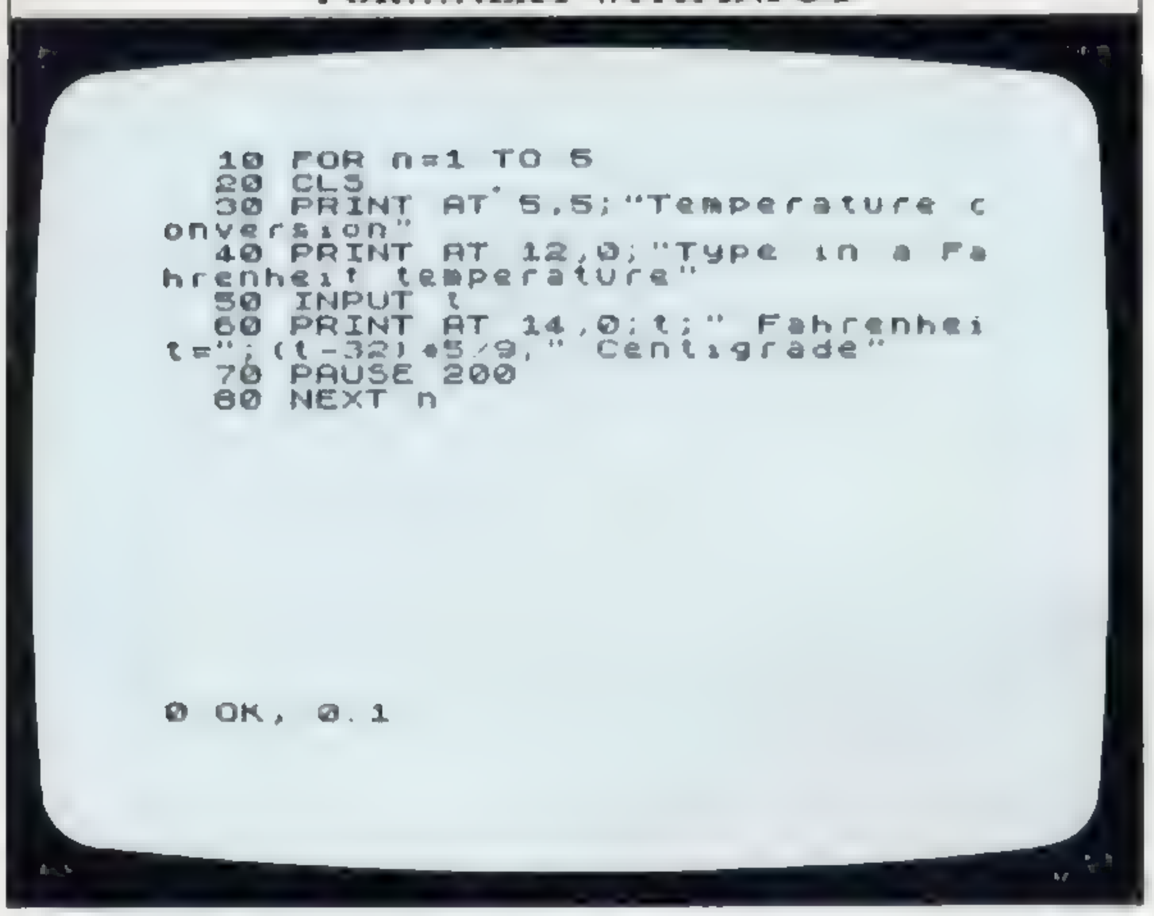


The FOR … NEXT loop both improves the program and shortens it by one line. Note that you don't have to include LET x= or add 1 to x on each loop of the program now, because FOR … NEXT takes care of it automatically. It starts off by setting x equal to 1 and PRINTing x and x-squared. Line 40 asks for the next value of x and so the program re-starts from line 20, the

beginning of the FOR ... NEXT loop, and executes the intervening lines once more. This continues until x has a value of 21, the maximum set by line 20, and in this case, the program stops.

If necessary, the loop can be interrupted on each pass through to wait for new information. Try using INPUT in the middle of a FOR ... NEXT loop:

FOR...NEXT WITH INPUT



This program converts Fahrenheit temperatures into Centigrade. The FOR ... NEXT loop beginning at line 10 sets a limit of five calculations, after which you will have to RUN the program again. The INPUT statement at line 50 stops the program until you type in the Fahrenheit temperature you want to convert. Line 60 then does the calculation and PRINTs the result.

Slowing a loop down

One problem you may encounter when writing programs is that they often RUN too fast for you to be able to follow anything which is PRINTed on the screen. Line 70 in the temperature conversion program is used to deal with this problem. The command PAUSE stops the program temporarily so that the result of the conversion stays on screen long enough for you to read it before the next run through the loop begins and the screen is cleared. The length of this halt is set by the number following PAUSE. This number represents the length of the PAUSE in fiftieths of a second. Hence PAUSE 200 interrupts the program for 200/50ths or 4 seconds whereas PAUSE 0.5 is just 1/100th of a second.

How to round numbers off

The layout of the conversion display could be improved. It's fine as long as the result of the calculation is a whole number, but it rarely is. The more figures there are after the decimal point, the further "Centigrade" is pushed along the line until it splits and part of it ends up on the next line:

TEMPERATURE CONVERSION DISPLAY

To get around this problem, try replacing (t−32)*5/9 by INT ((t−32)*5/9+0.5). INT, short for INTeger, turns a decimal number into a whole number. If the result is 18.333333, for instance, adding INT changes that to 18. The number is more sensible, and the display looks much better:

ROUNDED-OFF CONVERSION DISPLAY

When you use INT, remember that it always rounds downwards to the next whole number. You may have wondered why the INT line has 0.5 before the last bracket. This is to ensure that INT always produces the nearest whole number. Adding 0.5 will achieve this. If you are confused by this, try keying in these two direct commands:

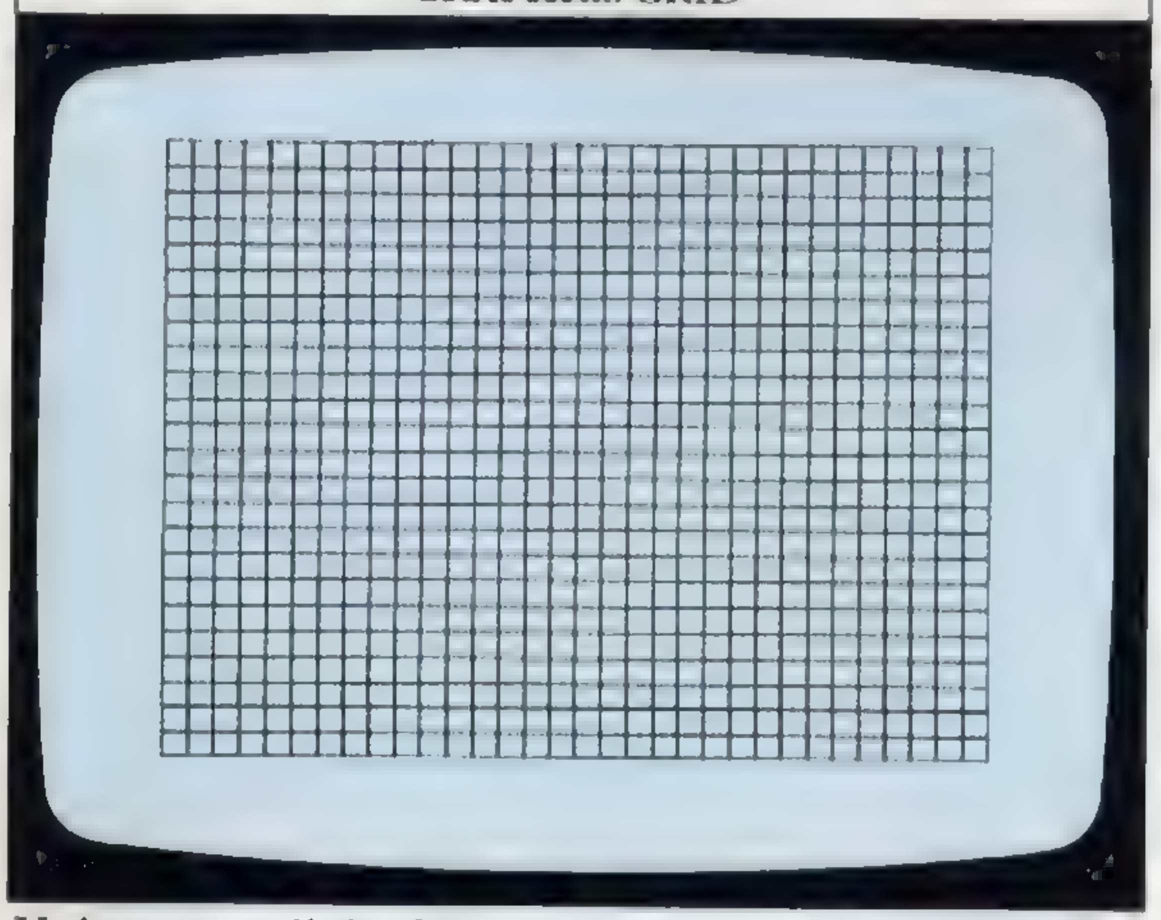
```
PRINT 3-1.1
PRINT INT (3-1.1)
```

The result of the first is 1.9, and the result of the second 1. But 1.9 is much nearer to 2 than 1. So, to compensate for this in the temperature conversion program, 0.5 is added before INT is used.

THE ELECTRONIC DRAWING-BOARD

The ZX Spectrum's BASIC includes several commands for drawing points and lines on the screen. If you want to draw a point or line, you must have some way of telling the computer where to start drawing. The screen is, therefore, divided up into a grid, made up of tiny dots. These dots are called picture elements (usually shortened to pixels) because the picture on the screen is made up from them. Each pixel is numbered – 0 to 255 across, and 0 to 175 up and down. The 0,0 position is at the bottom left corner of the screen. The co-ordinates of a point on the screen are always given as x,y, with x (the number of pixels across the screen from the left) first, followed by y (the number up from the bottom of the screen). On the screen below, each grid square is 8 pixels wide and 8 pixels high:

GRAPHICS GRID



Unless you tell the Spectrum otherwise, it assumes that you want to start drawing from 0,0 (also called the "origin") or from the last position visited. You can make a point appear on the screen by using the Spectrum's PLOT command. To make a black dot appear at the centre of the screen, type:

```
PLOT 128,88
```

You can check these co-ordinates on page 61.

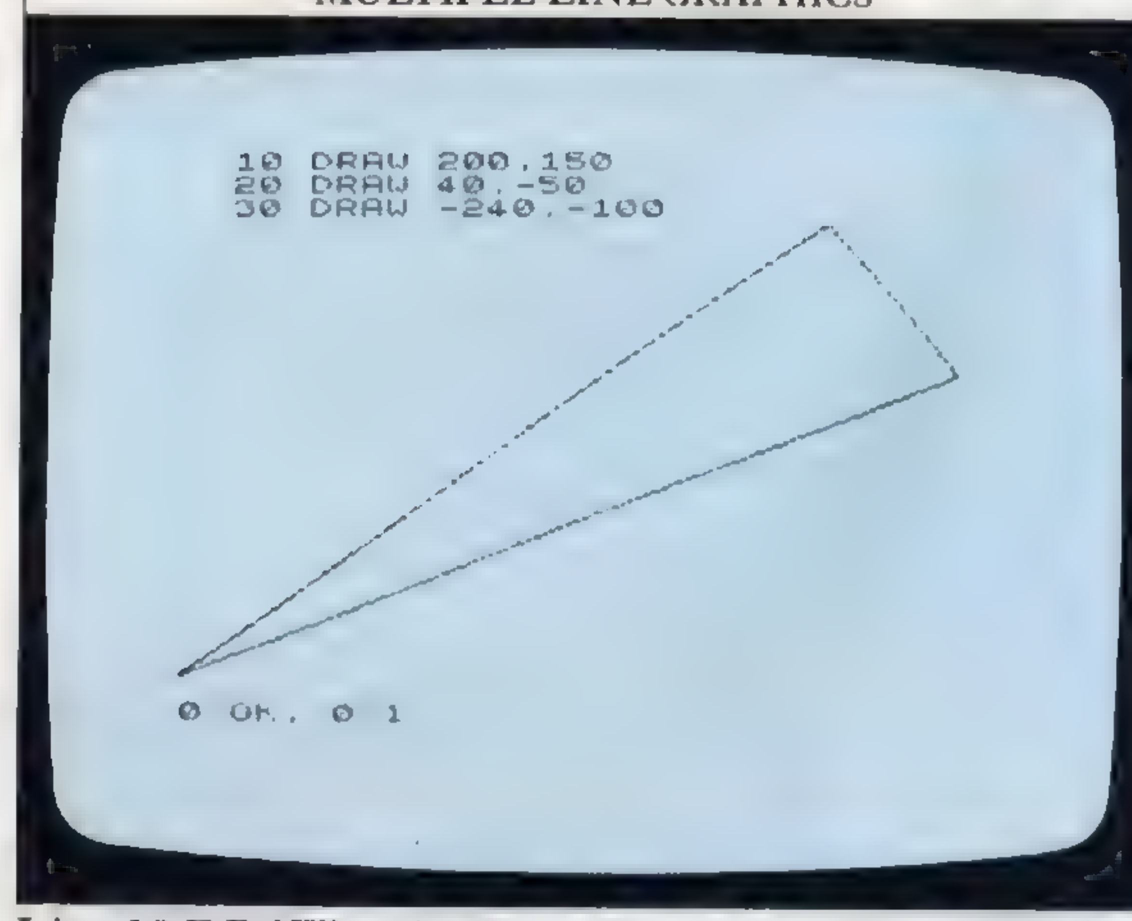
How to draw lines

Lines are drawn on the screen using the command DRAW. For instance:

```
DRAW 128,88
```

DRAWs a line from the bottom left corner of the screen, the origin, to a point in the middle of the screen. If you then instruct the computer to DRAW a second line, it will automatically start DRAWing from wherever the last line ended. You can use this to your advantage to produce simple, straight-line shapes very easily. Here is a LISTed program that produces three lines, together with the shape it DRAWs:

MULTIPLE LINE GRAPHICS

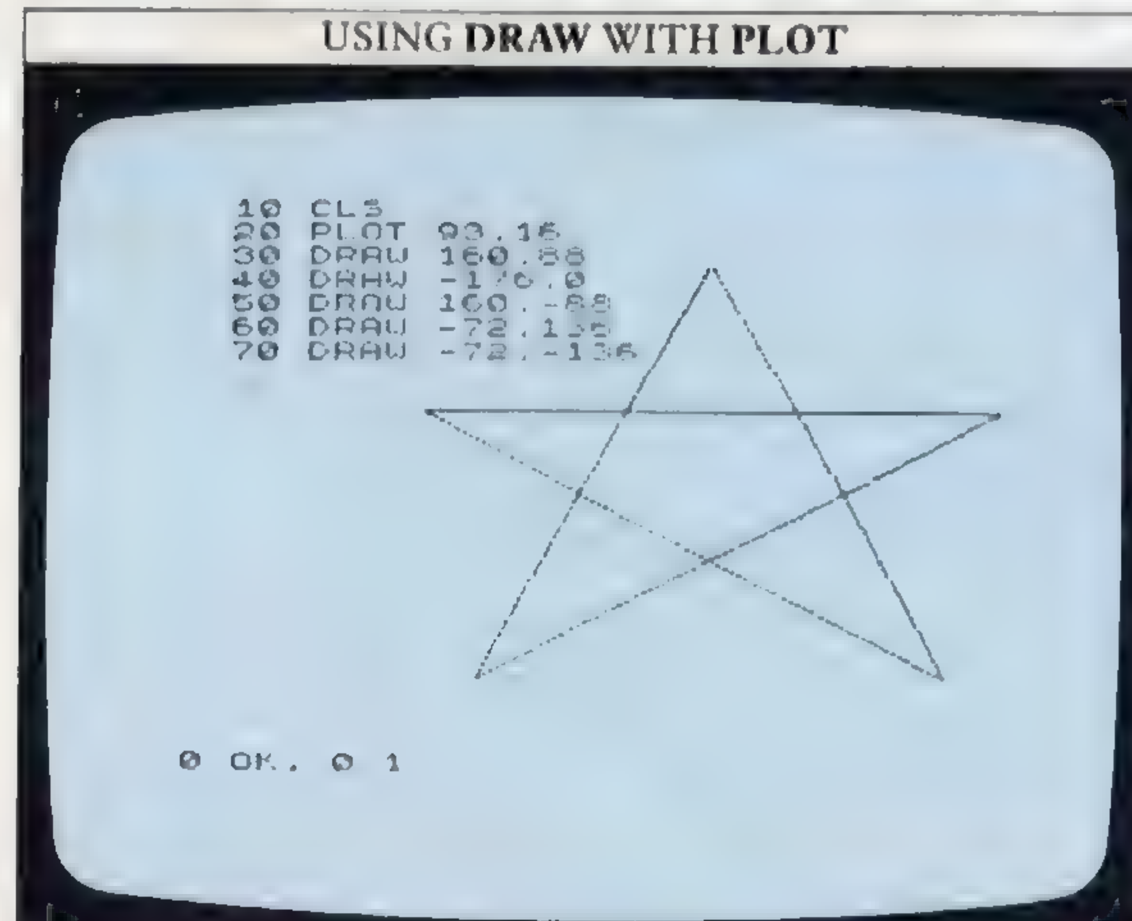


Line 10 DRAWs a line from the origin at 0,0 to 200,150. The next line is to be DRAWn from 200,150 to 240,100. The changes in the x and y co-ordinates are 40 and −50 respectively, so line 20 is therefore DRAW 40, −50. Finally, line 30 DRAWs the third side of the triangle from 240,100 back to the origin.

Picking a starting point

To DRAW a shape in the middle of the screen, you must tell the computer that you want to start DRAWing from somewhere other than the origin at 0,0. The PLOT command takes care of that. First, PLOT a point where you want to begin DRAWing. Now, the next line DRAWn will begin from that point:

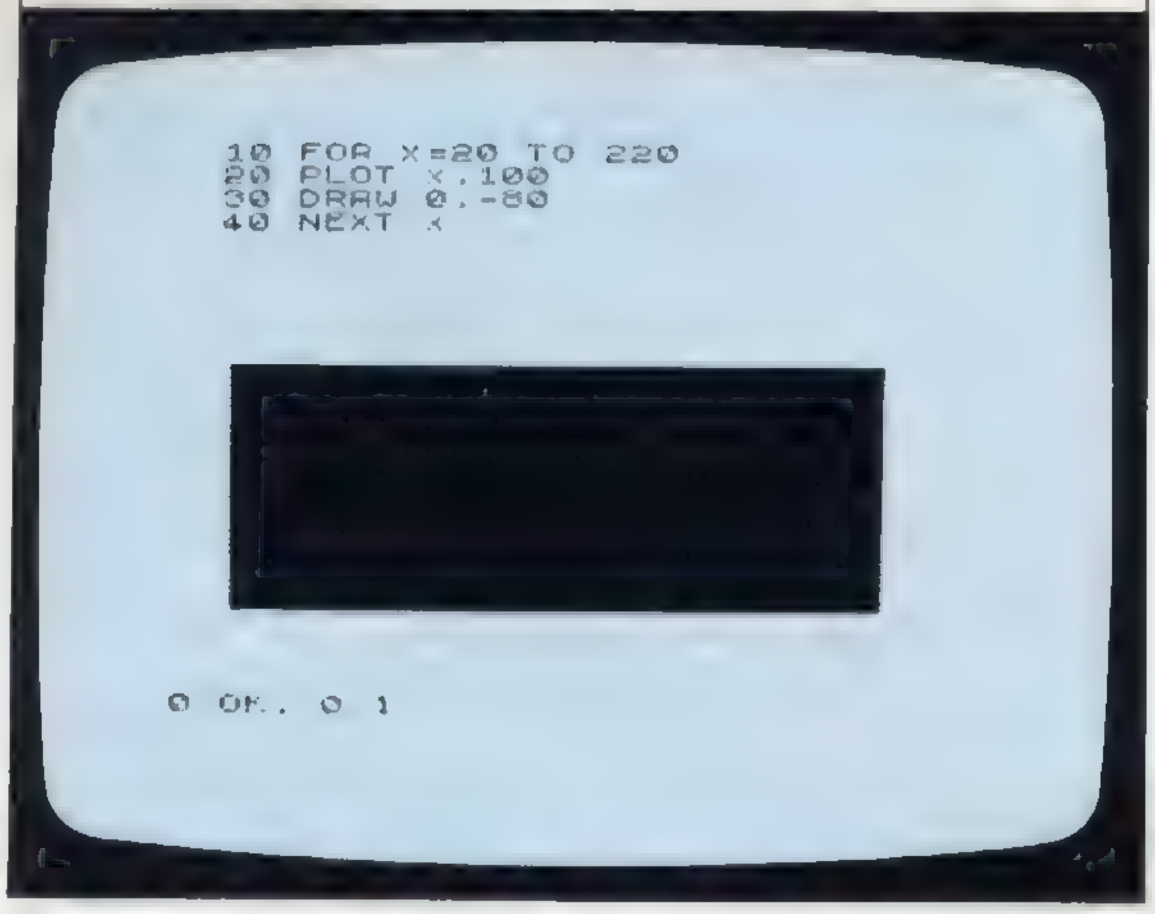
USING **DRAW** WITH **PLOT**



How to fill in shapes

It is now a simple matter to fill in these line drawings to produce solid black figures:

SOLID RECTANGLES WITH **FOR...NEXT**



This repeatedly DRAWs lines from the top of the rectangle down to the bottom, gradually working from left to right, generating a solid black rectangle. A solid triangle is produced in a different way. The lines are all DRAWn from a single point, one apex of the triangle:

SOLID TRIANGLES WITH **FOR...NEXT**

Now let's try something a little more ambitious. In addition to points and lines, the Spectrum can generate a number of graphics characters that are permanently stored in its memory. You can see them printed on the tops of keys 1 to 8. The largest is a square the size of one character, while the others display halves, quarters and other fractions of squares. Using these characters is a simple way of generating graphics, but it produces only coarse images, and, since each character can occupy only character positions, only jerky movements are possible when you come on to animation.

Selecting them involves switching to the graphics cursor. For instance, to make a black square type:

PRINT "■"

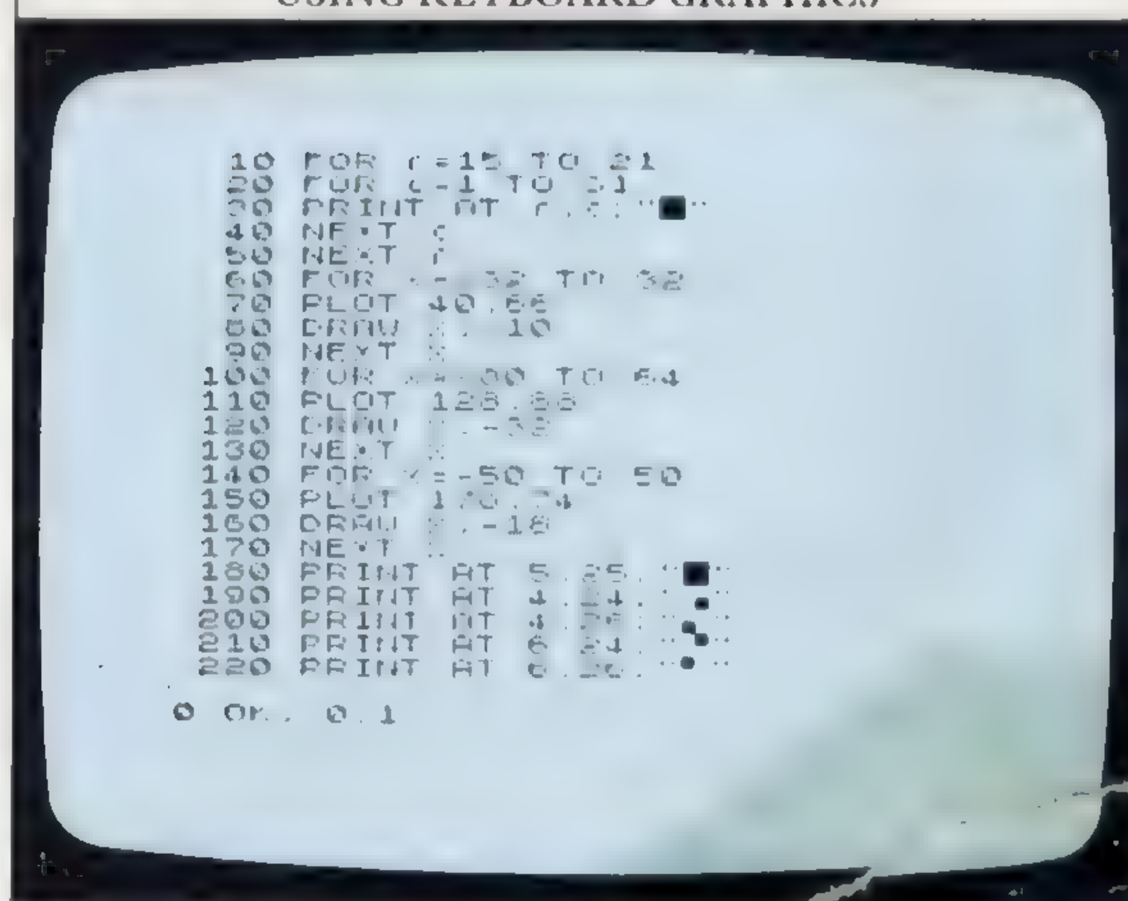
To produce this, before pressing key 8, switch to the graphics cursor by pressing CAPS SHIFT and 9. Now, holding CAPS SHIFT down, press key 8 to produce the character. Before continuing, press CAPS SHIFT and 9 again to remove the graphics cursor. You can produce the inverse (negative) of any keyboard graphic by not using CAPS SHIFT again when pressing the symbol key. So, to summarize, the sequence of key presses needed to produce the black square is:

CAPS SHIFT and 9
CAPS SHIFT and 8
CAPS SHIFT and 9

DRAWing a simple landscape

The following program shows you how you can use these graphics symbols together with the PLOT and DRAW commands to produce a basic landscape:

USING KEYBOARD GRAPHICS



DESIGNING YOUR OWN CHARACTERS

You can make up almost any shape of graphics character by using PLOT and DRAW as described on pages 28–29. However, it is much more convenient if you store your own graphics characters in the computer's memory in the same way as it stores all of its own keyboard characters. You can then treat a graphic symbol – a rocket ship or a human figure – as a single character, instead of having to RUN a special program to DRAW and PLOT the symbol each time you need it. The Spectrum allows you to program the A to U keys with any graphics symbols of your choice.

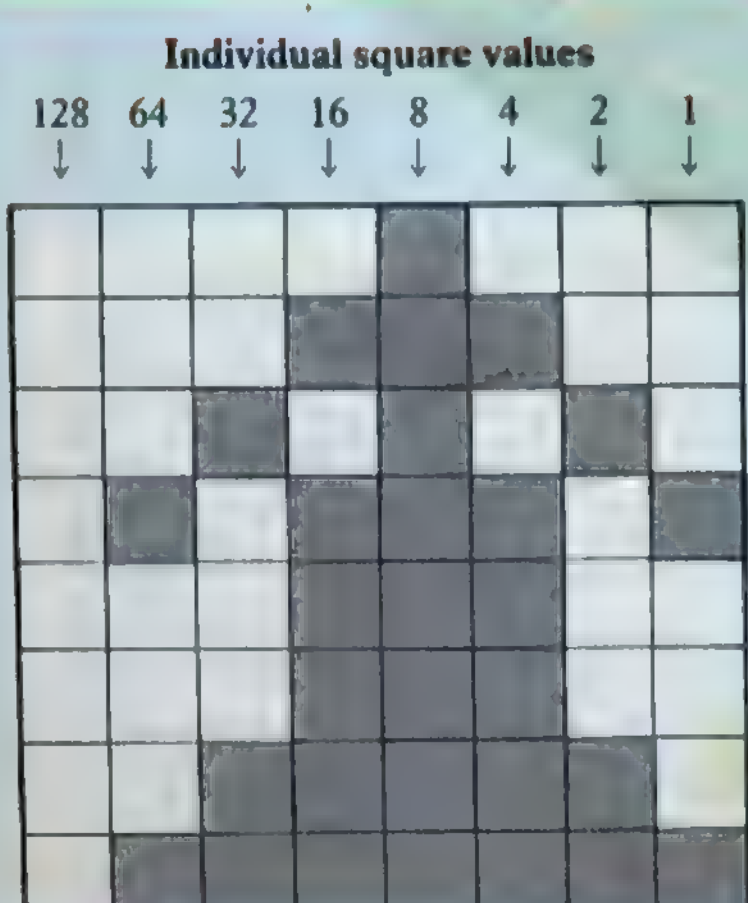
How to use a character grid

Say you want to put a small rocket on the screen for a space game. The shape will be shown on the screen as a pattern of dots on an 8×8 grid, so draw one out on an 8×8 grid on a piece of paper, as on the diagram below, or use the grid on page 61.

Draw the shape by blacking in whole squares. When you have done this, add up the numerical values of the squares in each row. On the top row of the example, only one square, the one with 8 above it, is blacked in, so the total for the top row is 8. In the next row, the squares labelled 4, 8 and 16 are black, so the total for row 2 is 28, and so on. These totals can then be fed into the computer to reprogram one of its keys.

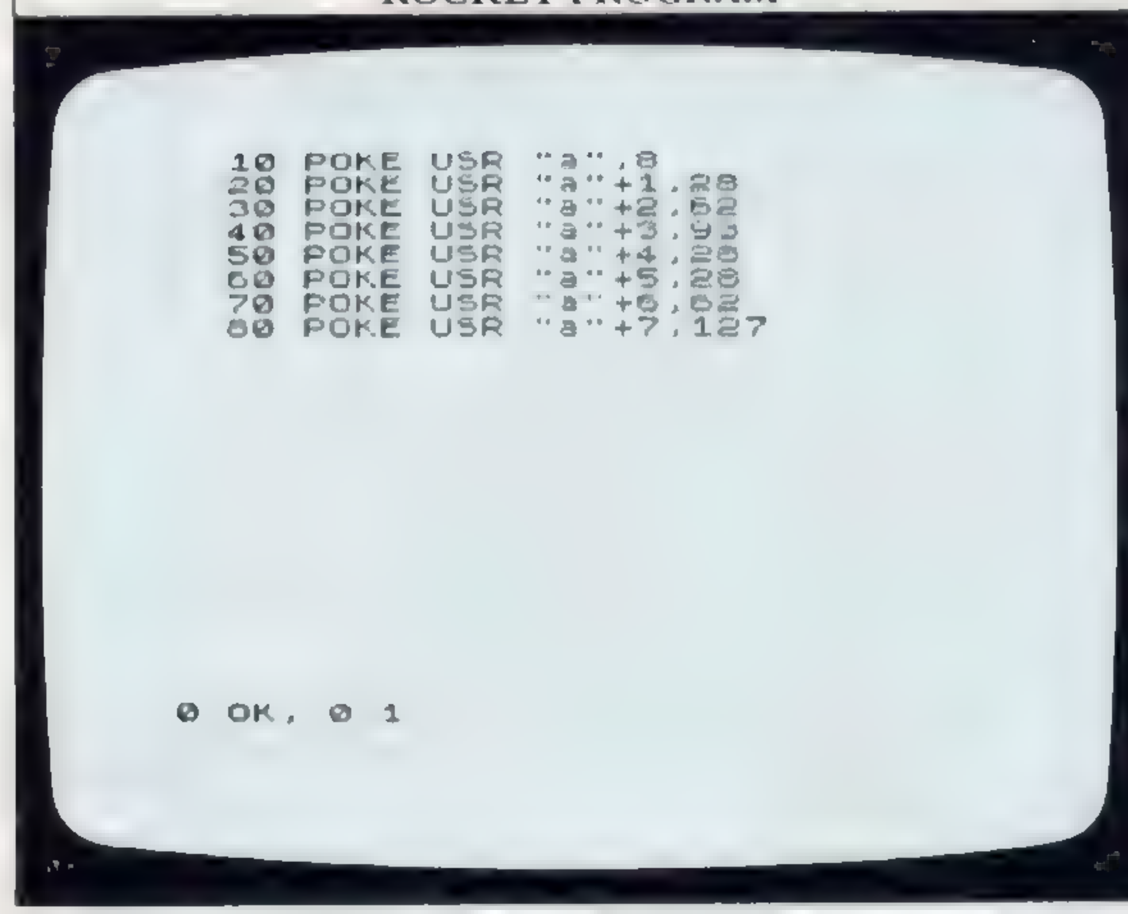
POKE is a command which allows you to put information directly into the computer's memory. USR"a" in the following program tells the computer that you want to recover your character by pressing the "a" key. (Using a capital A would work equally well). The number following (+1, +2, etc) identifies which row of the 8×8 grid the final number refers to. Eight lines of the program are necessary to install the new character, working from "a" through to "a"+7. Here is the rocket and the program that stores it:

SINGLE GRID CHARACTER



Row Totals	
8	= 8
16+8+4	= 28
32+8+2	= 42
64+16+8+4+1	= 93
16+8+4	= 28
16+8+4	= 28
32+16+8+4+2	= 62
64+32+16+8 +4+2+1	=127

ROCKET PROGRAM



Now you need a way of getting your character back out of memory and onto the screen. The PRINT statement is used. To PRINT the newly programmed character in the middle of the screen add this line to the above program, first changing to the graphics cursor, and then repeatedly pressing the A key:

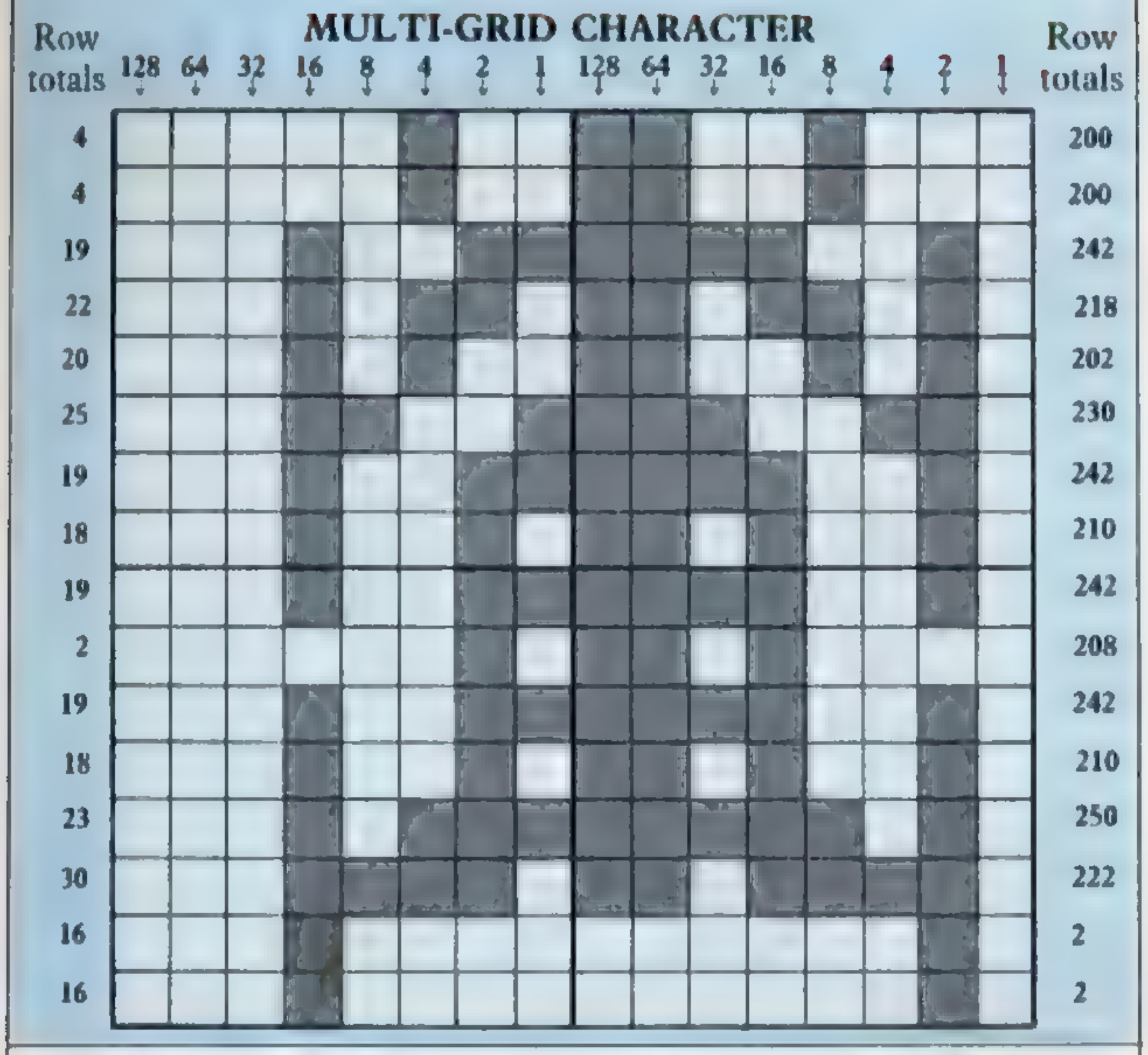
```
90 PRINT AT 15,8;"AAAAAAAAAAAA"
```

ROCKET DISPLAY

By switching to the graphics cursor, pressing the A key repeatedly, and then removing the graphics cursor again as described on page 29, a row of the character you have designed is entered in the program line. Then just RUN the program.

How to add characters together

The first thing you will notice is that the rockets are extremely small. Have a try at something larger:



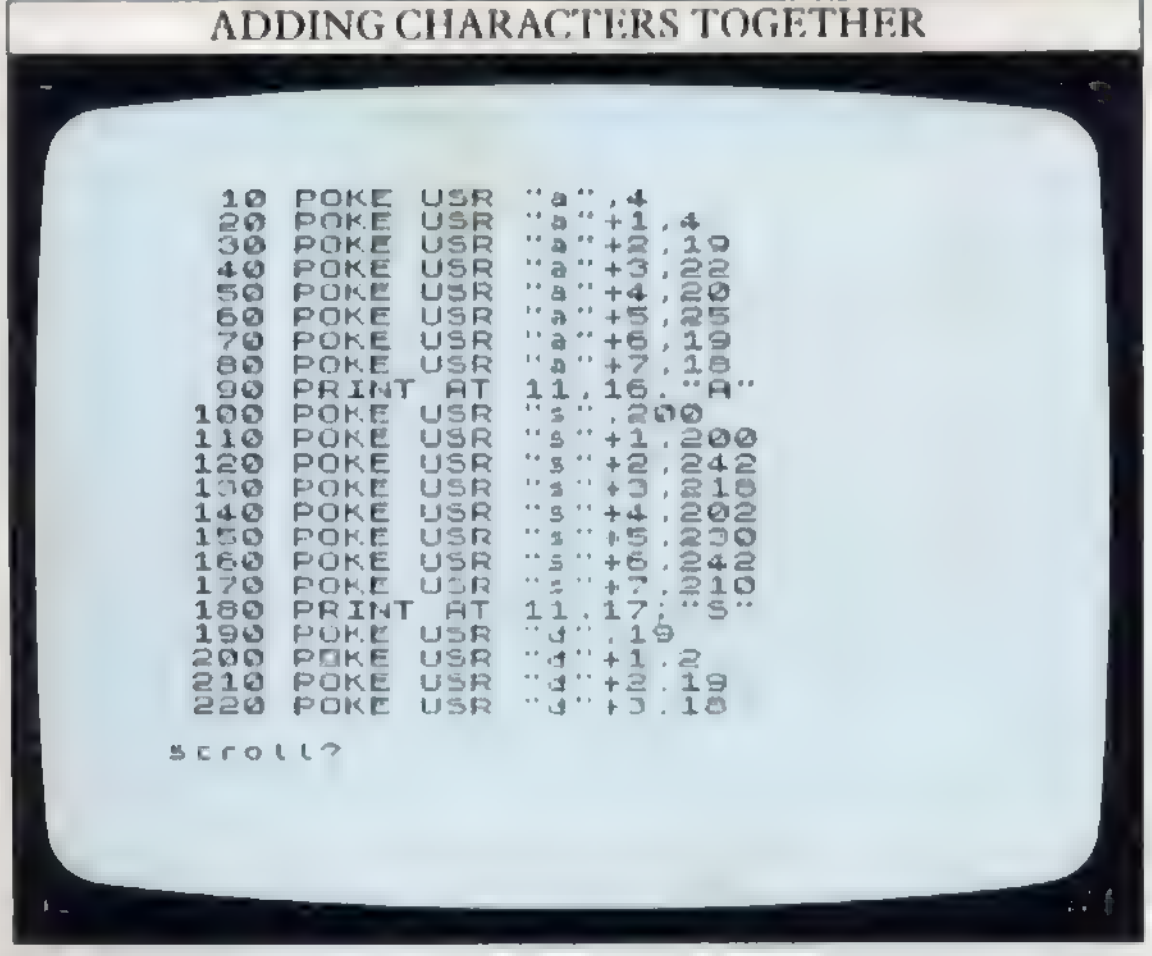
Although user-defined characters are based on an 8×8 grid, there's no reason why your character should not cover more than one grid.

Again, key in the totals representing each row in each 8×8 grid. Each grid must be labelled with a different keyboard letter to identify it. In the program that follows the new rocket, they are labelled with the letters a, s, d, and f.

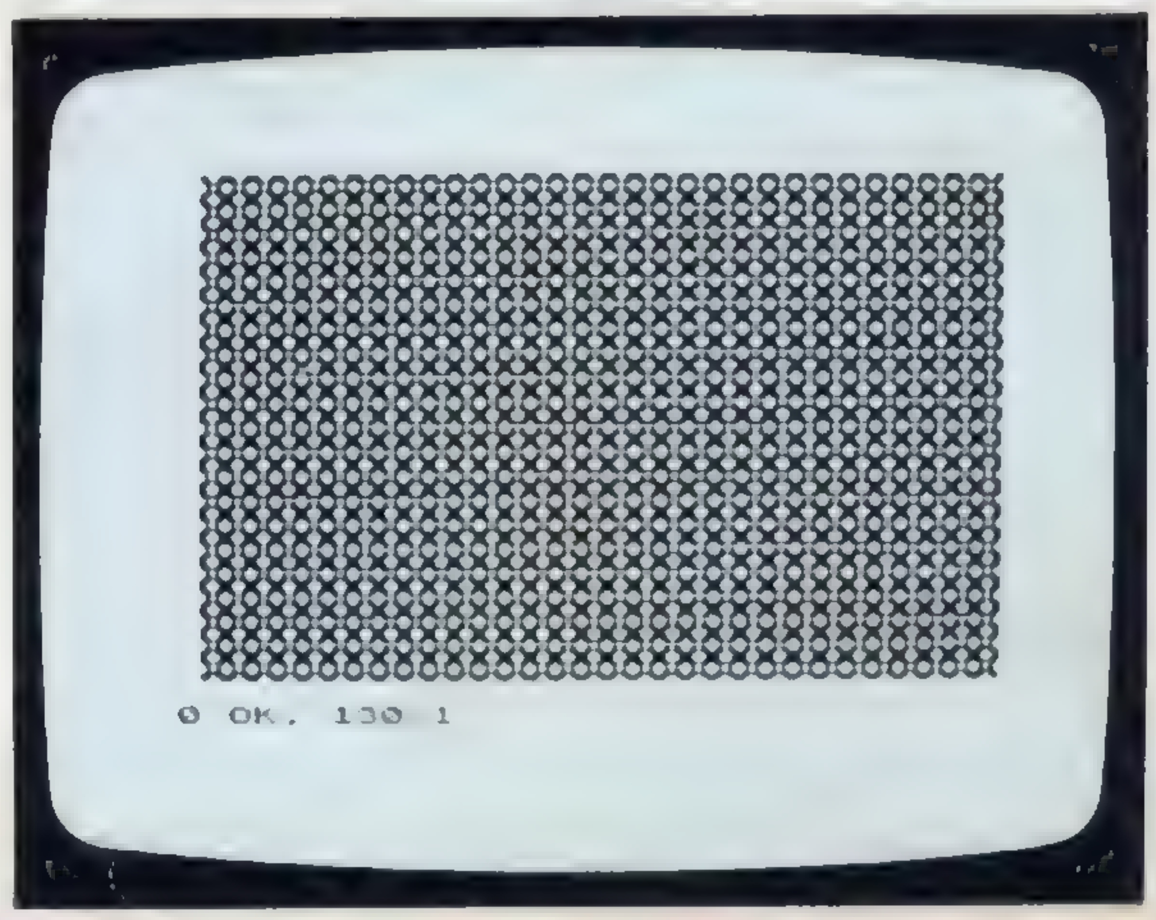
Screen patterns with POKE USR

With user-defined graphics it is possible to use your own characters to make a pattern by PRINTing them all over the screen. In this program, try filling in the zeros in lines 10–80 to define a character (the grid on page 61 will help you with this). Instead of giving the character a fixed position such as in the above example, the program will PRINT the character AT r,c where r (the row number) and c (the column number) are constantly changing. Here is the program and a display made by replacing all the zeros in lines 10–80 with 195,231,126,36,36,126,231,195:

ADDING CHARACTERS TOGETHER



PATTERN GENERATOR PROGRAM



ANIMATION

Once you have learned how to create a character and make it appear anywhere on the screen using PRINT AT, you can then attempt some simple animation by using the same method to make the character move. First, you need a program that produces a character:

STATIC CHARACTER

This program creates an "alien" with two graphics symbols by reprogramming the Q and R keys. Together, the symbols make up an alien in the middle of the screen. Line 170 tells the computer to PRINT the top half of the alien AT 11,15, while line 180 uses AT to put the bottom half in the right place.

Now you can try making it move from one side of the screen to the other. You will be relieved to know that lines 10 to 160 inclusive remain exactly the same, so don't press NEW before making the following changes or you will have to type in the whole of the program again. There's no need to delete lines 170 or 180, because the new lines will replace them.

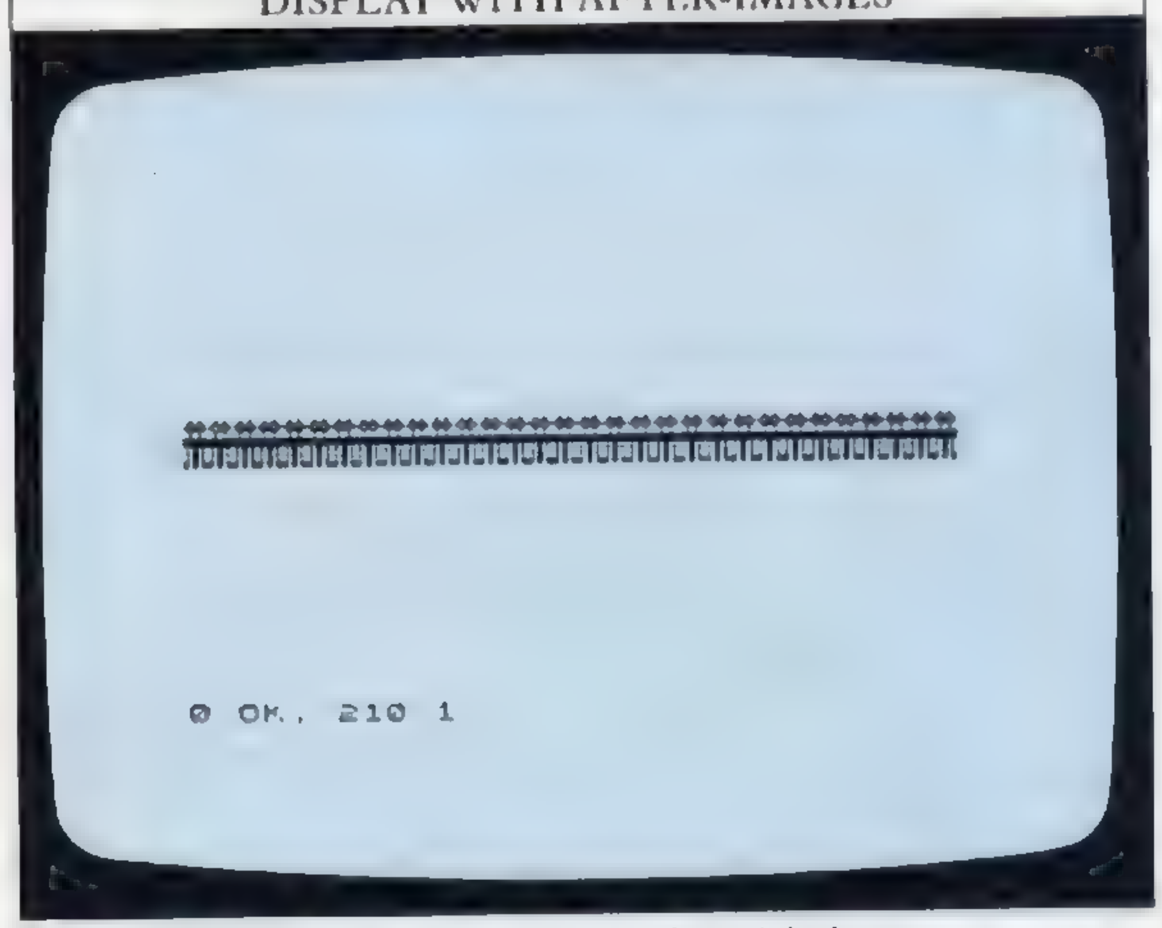
These lines and changes will start the alien moving (the graphics symbols will automatically replace Q and R if you have RUN the original program):

ANIMATION LINES

How to remove after-images and control speed

When you RUN the modified program, the first thing you will notice is that it doesn't do exactly what you want it to. The Spectrum moves the alien from left to right, but as it moves the alien, it doesn't remove the old images. The result is a long line of characters:

DISPLAY WITH AFTER-IMAGES



The problem is that you haven't told the computer to remove the old unwanted images. Doing that is quite easy – just PRINT a blank space after each of the characters with these lines:

```
205 PRINT AT r,c-1;" "
206 PRINT AT r+1,c-1;" "
```

The alien now moves from left to right without leaving a trail. But it's very fast. How can you scale down the speed? Put in a time delay:

207 PAUSE 2

Now the alien takes longer to reach the right side of the screen. You can adjust its speed by altering the PAUSE. This screen shows an impression of the movement (the after-images will not actually appear on your screen):

ANIMATION WITH DELETIONS

You will find that the slower the speed, the more clearly the alien appears on the screen. Flickering occurs when a symbol is moving quickly, or if it is made up of a number of characters. With a two-character symbol like this, there is a slight time delay between the computer PRINTing the first and the second character. If you use more than two characters to make up a symbol, the flickering as the symbol moves will be more noticeable.

Movement up and down the screen

You can move something down the screen from top to bottom, or vice versa, equally easily. Instead of varying the horizontal position (shown by the changing variable c in the alien program), you change the vertical position by the same method.

The next program stores a symbol representing a small rocket under the A key, and then makes it fall down the screen. The same sort of FOR ... NEXT loop as was used in the alien animation is used here at line 140. (FOR...NEXT and other program loops are explained on pages 26–27.) The r (row) variable increases so that the rocket is PRINTed at progressively lower positions.

To make sure that the rocket does not leave a trail, the program PRINTs a blank space behind it. Because this program uses a symbol only one character wide, only one blank space is needed. To give the rocket something to hit, lines 90 to 120 PRINT a simple landscape with a horizon, just like on page 29. Line 190, which is only carried out after the FOR ... NEXT loop has been completed, PRINTs five graphics characters (ones found on the number 6 key). This line produces the impression of the rocket's impact:

ARROWHEAD FALL PROGRAM

INTRODUCING COLOUR

The Spectrum's television display is not limited to black letters and symbols on a white background. It can actually PRINT in any of eight colours (including black and white) and you can insert these colours in your programs. Each colour is identified by a number.

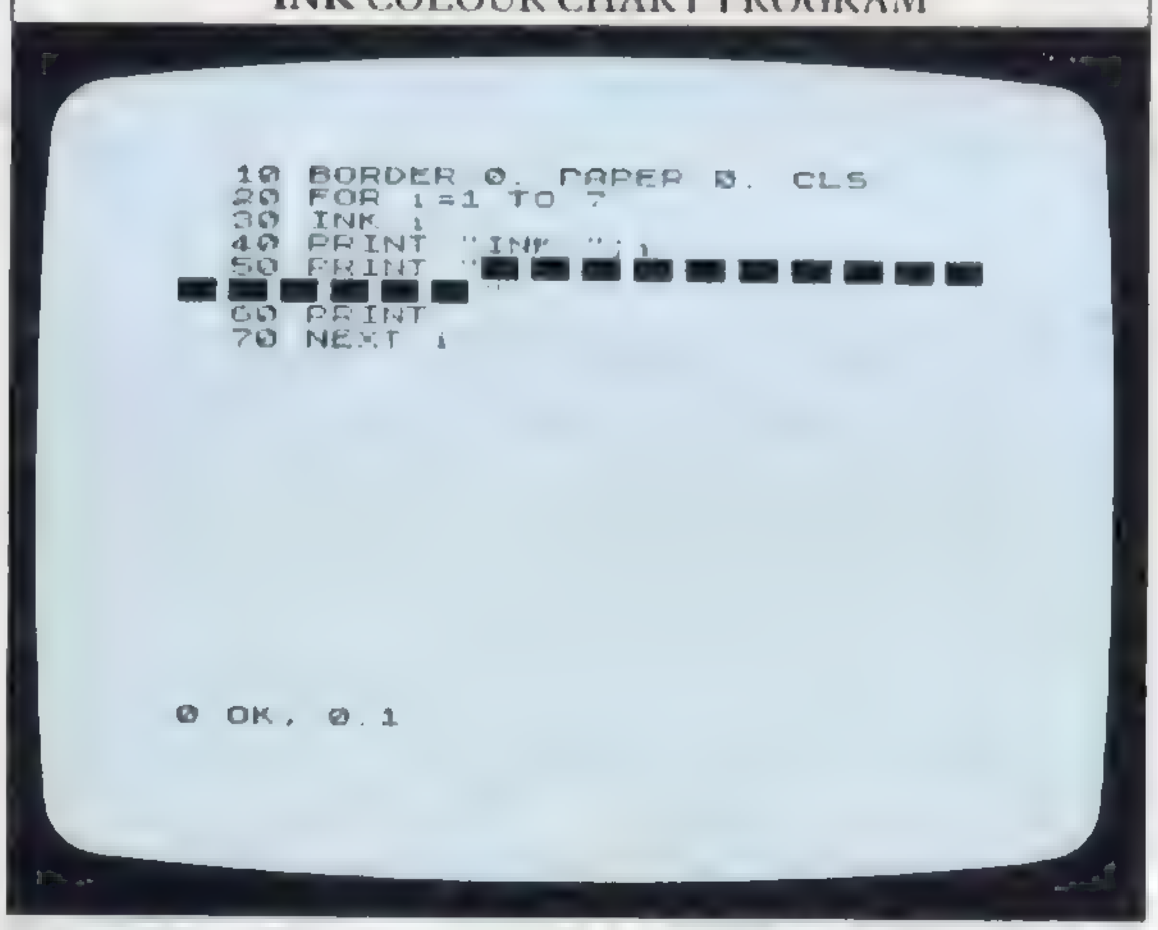
If you look at the number keys, you will see that keys 1–7 and 0 have a colour printed above them. To produce a colour on the screen, you need to use one of these colour codes together with a colour command.

Using colour commands

The Spectrum has three ways in which you can control colour. INK (on the X key) controls the colour of text PRINTed on the screen. PAPER (on the C key) selects the colour of the background, and BORDER (on the B key) controls the colour around the edge of the screen. To select a colour, you just have to key in one of these commands followed by a colour number.

To see what colours INK produces, key in this colour chart program:

INK COLOUR CHART PROGRAM



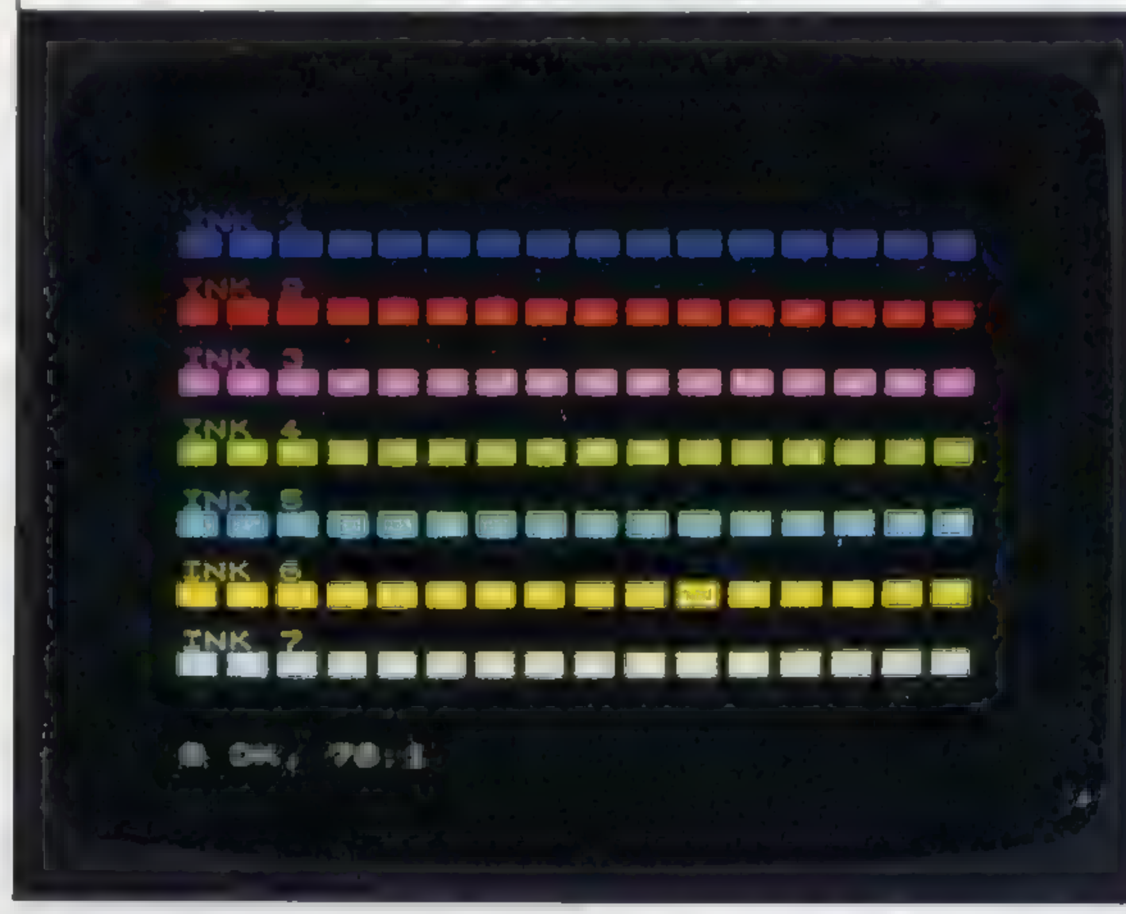
Lines 20 to 70 contain a FOR ... NEXT loop which PRINTs a row of graphics characters (on the number keys) in each of the INK colours from 1 to 7. The program PRINTs all the colours except black:

SPECTRUM COLOUR NUMBERS

Each colour command is followed by one colour number.

Number	Colour
0	Black
1	Blue
2	Red
3	Magenta
4	Green
5	Cyan
6	Yellow
7	White

INK COLOUR CHART

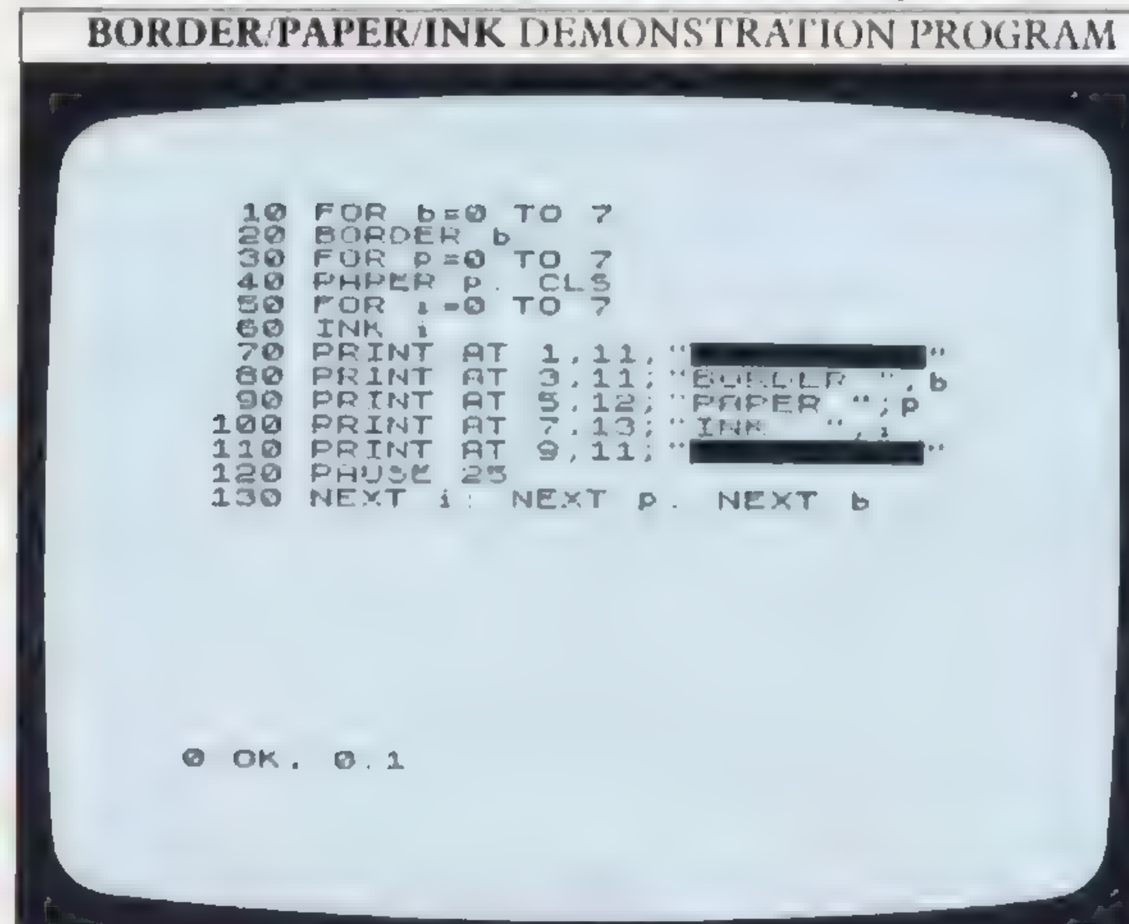


Line 10 makes the PAPER and BORDER areas turn black. You will notice that the line contains the command CLS. This is to ensure that the PAPER area is black before any text is PRINTed. Normally, the PAPER command only produces a coloured background immediately behind each PRINTed character. To colour the whole PAPER area, you must follow the PAPER command with CLS.

Changing INK, PAPER and BORDER

The next step is to see what happens when you use different INKs together with changing PAPER and BORDER commands. The following program does just this; it contains three FOR ... NEXT loops, one to control each of the colour commands. If you RUN it, you will see all the possible combinations of INK, PAPER and BORDER (there are 512 altogether!):

BORDER/PAPER/INK DEMONSTRATION PROGRAM



BORDER/PAPER/INK DEMONSTRATION DISPLAYS

These two displays are produced by keying in the demonstration program. When you are writing your own program listings, you can work with any INK, PAPER and BORDER colours just by keying a line in (without a line number) before you start your listing. White on black, which produces an easy-to-read display, can be produced by keying in:

INK 7:PAPER 0:BORDER 0:CLS

Pressing NEW will reset the computer.

Improving the picture

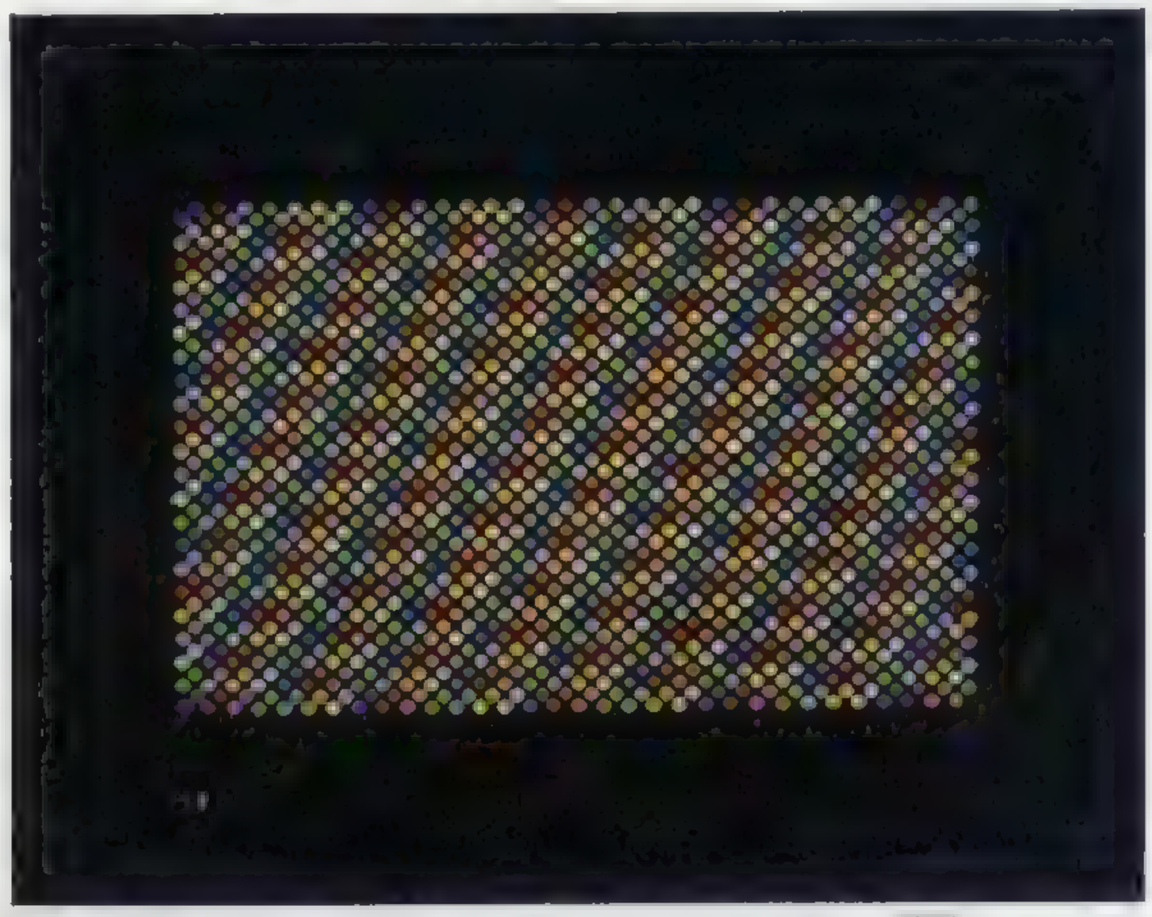
If you are disappointed with the television picture your Spectrum is producing, you may be able to improve it by going through some simple checks. Make sure that the television is properly tuned into the computer's output signal. The tuning setting of both may drift from time to time, so it is a good idea to check your television tuning periodically.

If the picture is covered by a herringbone pattern, and you have checked that the television is properly tuned, make sure that there is nothing nearby that might be interfering with the computer's signal – a video recorder or another television set for example. Finally, the more colourful and brilliant a picture is, the more distorted it will seem to be. Try reducing colour, brightness or contrast to improve this.

Colouring user-defined characters

If you define your own character with POKE USR (as on pages 30–31) you can PRINT this in any INK colour. Here is a program which produces a character – a pair of small stars – and which PRINTs it all over the screen in INK colours from 1 to 7:

COLOUR STARS PROGRAM



Experimenting with colour is simple. When you become more adept at it you can begin to create new colours by optical effects. For instance make up user-defined characters from a grid of dots of which every other dot is a second foreground colour. Now, colour the background in a contrasting colour. Red dots on a blue background for example will mix to appear purple, while red and yellow will appear orange.

COLOUR GRAPHICS

Colour can be added to Spectrum graphics using the same commands and techniques as are used to colour text and user-defined characters. You don't need to learn special commands, nor to write a long program to produce simple colour graphics. Here, for example, is the arrowhead fall program that you used for animation on page 33. If you type in the program again, you can then add some colour to it:

ARROWHEAD FALL PROGRAM

```
  10 POKE USR "a",255
  20 POKE USR "a"+1,255
  30 POKE USR "a"+2,126
  40 POKE USR "a"+3,126
  50 POKE USR "a"+4,60
  60 POKE USR "a"+5,60
  70 POKE USR "a"+6,24
  80 POKE USR "a"+7,24
  90 FOR r=15 TO 21
 100 FOR c=0 TO 31
 110 PRINT AT r,c;"■";
 120 NEXT c: NEXT r
 130 LET c=10
 140 FOR r=0 TO 14
 150 PRINT AT r,c;"A"
 160 PRINT AT r-1,c;" "
 170 PAUSE 5
 180 NEXT r
 190 PRINT AT r,c-2;"▚▚▚▚▚"

0 OK, 0:1
```

Programming graphics colour

Now key in an extra line to the program:

85 BORDER 2:PAPER 1:INK 4:CLS

If you look at the colour codes on page 34, you should be able to tell what effect the new line will have. Now RUN the program. Here is the adapted listing containing the new line:

COLOUR ARROWHEAD FALL PROGRAM

```
  10 POKE USR "a",255
  20 POKE USR "a"+1,255
  30 POKE USR "a"+2,126
  40 POKE USR "a"+3,126
  50 POKE USR "a"+4,60
  60 POKE USR "a"+5,60
  70 POKE USR "a"+6,24
  80 POKE USR "a"+7,24
  85 BORDER 2: PAPER 1: INK 4: C
LS
  90 FOR r=15 TO 21
 100 FOR c=0 TO 31
 110 PRINT AT r,c;"■";
 120 NEXT c: NEXT r
 130 LET c=10
 140 FOR r=0 TO 14
 150 PRINT AT r,c;"A"
 160 PRINT AT r-1,c;" "
 170 PAUSE 5
 180 NEXT r
 190 PRINT AT r,c-2;"▚▚▚▚▚"

0 OK, 0:1
```

You should find that the adapted program now produces a red arrowhead, a blue sky, green ground and red BORDER. When the arrowhead reaches the ground, the graphics characters that show the impact are now PRINTed in green against blue – because line 190 is also affected by the INK and PAPER commands in line 85. Here is the coloured display that the adapted program produces after the arrowhead has completed its fall from the top of the PAPER area:

COLOUR ARROWHEAD FALL DISPLAY

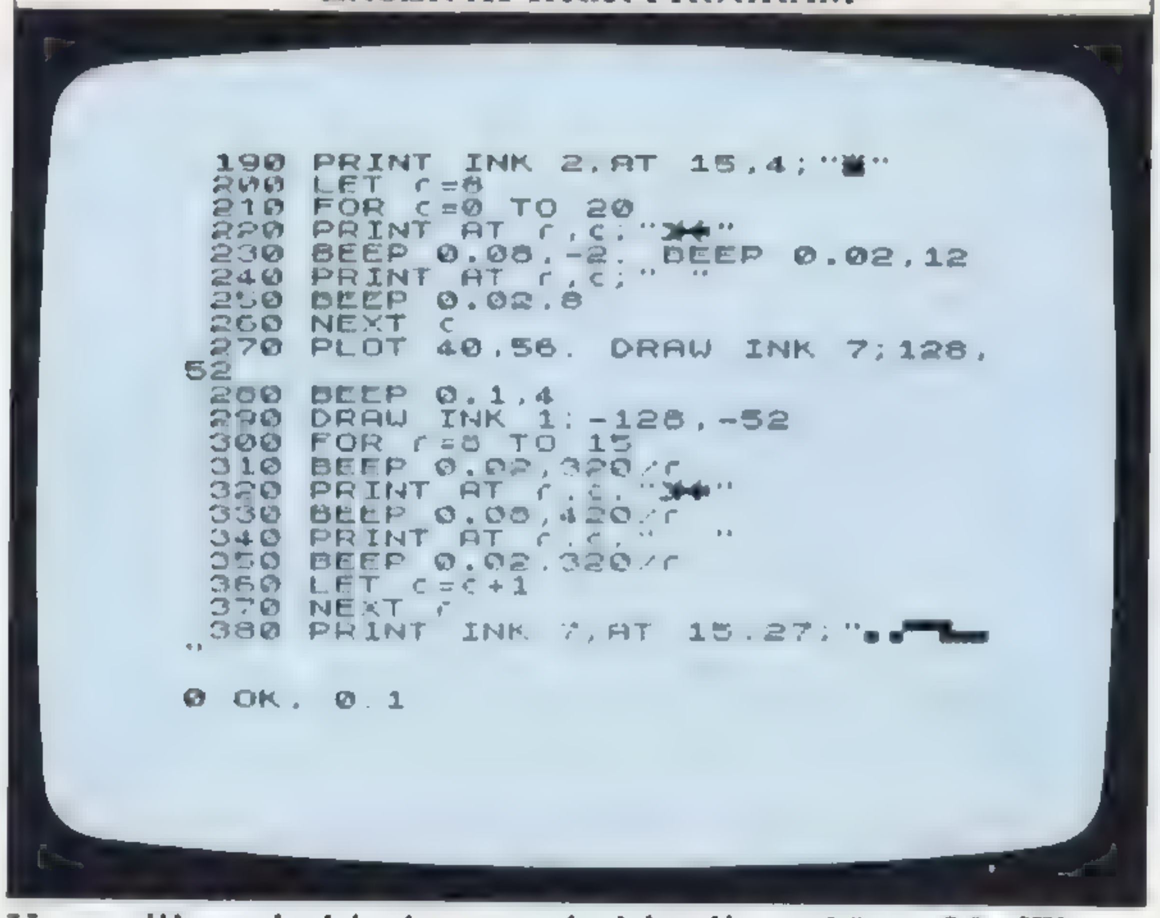
Animated action in colour

You now have the basic expertise needed to write an animated colour graphics program. As an example of some simple colour graphics, here is a program which brings together everything that you have learned so far. It is built up from a number of separate blocks or "modules", most of which you should be able to follow:

LASER ATTACK PROGRAM

```
  10 DATA 2,244,46,31,31,46,244,
2
  20 DATA 8,16,56,254,254,56,16,
8
  30 DATA 146,84,84,254,124,254,
124,254
  40 FOR n=0 TO 7: READ x
  50 POKE USR "a"+n,x
  60 NEXT n
  70 FOR n=0 TO 7: READ x
  80 POKE USR "s"+n,x
  90 NEXT n
 100 FOR n=0 TO 7: READ x
 110 POKE USR "d"+n,x
 120 NEXT n
 130 BORDER 0: PAPER 1: INK 6: C
LS
 140 FOR r=16 TO 21
 150 FOR c=0 TO 31
 160 PRINT INK 4;AT r,c;"■"
 170 BEEP 0.01,14
 180 NEXT c: NEXT r

scroll?
```

LASER ATTACK PROGRAM



You will probably be puzzled by lines 10 to 30. These lines are actually part of a quick way of producing user-defined characters. If each of the a, s and d keys were reprogrammed by the method we've been using up to now, it would take 24 lines – 8 lines of POKE USR statements for each letter. Using this new method, each key can be reprogrammed in a maximum of four lines, with a list of grid totals stored after each DATA command. The program uses three characters; two to make the front and back of a rocket, and another one to make a laser base.

The READ...DATA routine, explained on pages 50–53, is a quick method of assembling all the data you need to make up user-defined graphics. It enables you to put all the data into one statement, instead of using the POKE USR routine line by line.

Lines 40 to 120 then take the information that is stored in lines 10 to 30, and turn it into the characters themselves. The green ground is produced by lines 140–180, which contain a double FOR ... NEXT loop that repeatedly PRINTs a green square across the bottom of the screen. Line 190 then PRINTs the laser base.

Lines 200 to 260 control the movement of the rocket, sending it across the screen at a fixed height. You can see it on the top screen of the three in the right-hand column. You probably will not need to be told that the BEEP command produces the sound of the rocket. (You will shortly be coming on to using this command both for sound effects and notes).

Lines 270 to 300 program the laser to fire by DRAWing a line that hits the rocket (see the centre screen on the right). Once this has happened, lines 300 to 370 send the rocket plunging to the ground, while the final line PRINTs its crumpled wreckage. (In the final screen on the right the rocket is shown without the deletions which actually occur).

When you type in this program, remember that all the graphics symbols are obtained by switching to the graphics cursor (CAPS SHIFT +9) and pressing the appropriate user defined key – a, s or d. Remember also to remove the graphics cursor again before continuing.

LASER ATTACK DISPLAYS

SPECIAL SCREEN TECHNIQUES 1

As you saw on pages 32–33, it is possible to animate characters simply by PRINTing, erasing and rePRINTing the characters in a new position. If you want to take animation displays a step further, you can make use of two new Spectrum keywords INVERSE and OVER. These allow you to produce animation when the ordinary PRINTing technique will not work.

INVERSE is a command which switches over the INK and PAPER dot pattern on the screen, to give an effect which looks like the negative of a character. OVER lets you PRINT one character over another, so that the normal erasing that happens when you overPRINT does not take place.

INVERSE and OVER can be used to PRINT, PLOT or DRAW something over a background that itself has something DRAWn or PRINTed on it. This program shows what happens if you try to animate over a background without using these commands. It DRAWs a laser beam which fires across a grid:

LASER AND GRID PROGRAM

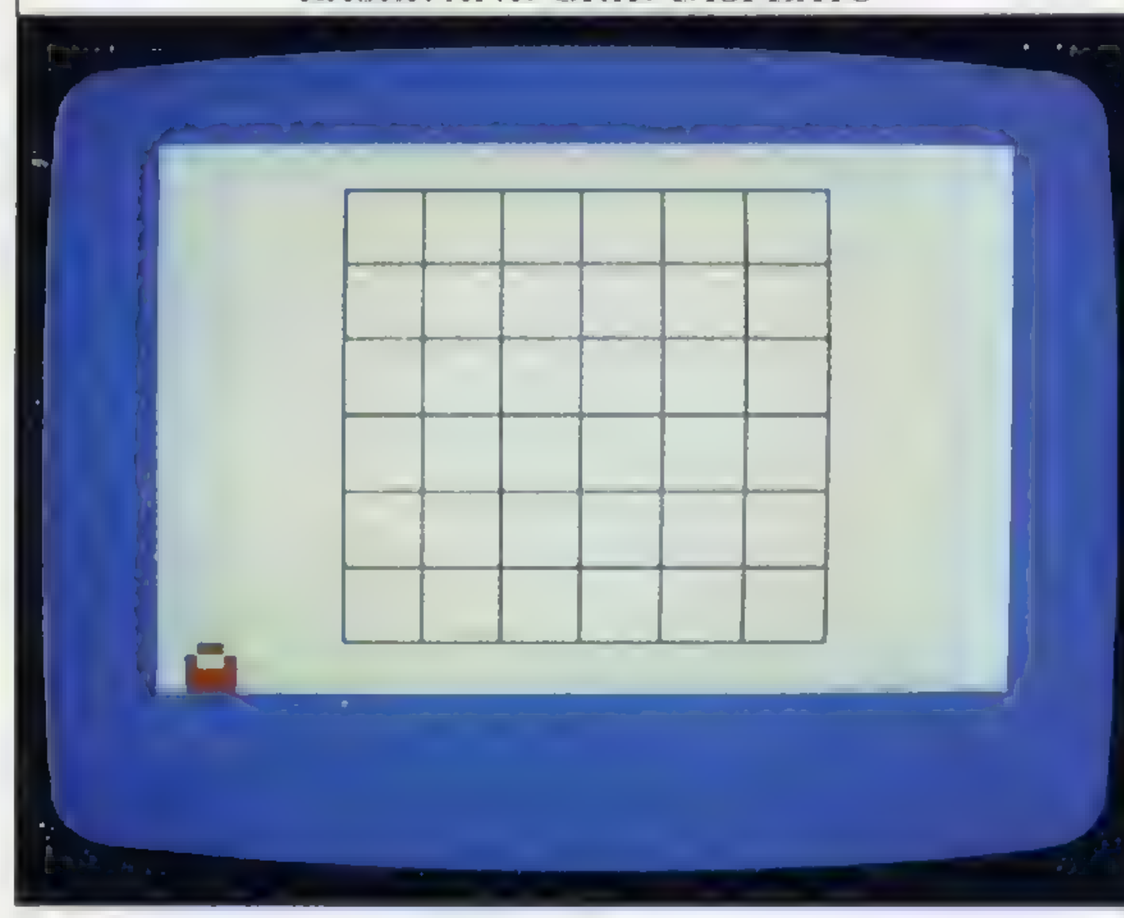
```
  10 BORDER 1. PAPER 7. INK 0. C
LS
  20 FOR y=16 TO 160 STEP 24
  30 PLOT 56,y. DRAW 144,0
  40 PLOT y+40,16: DRAW 0,144
  50 NEXT y
  60 PRINT INK 2;AT 20,1;" "
  70 PRINT INK 2;AT 21,1;" "
  80 PLOT 16,16
  90 LET x=INT (RND*140). LET y=
INT (RND*160)
 100 DRAW INK 0:x,y
 110 DRAW INK 7:-x,-y
 120 GO TO 80

0 OK, 0.1
```

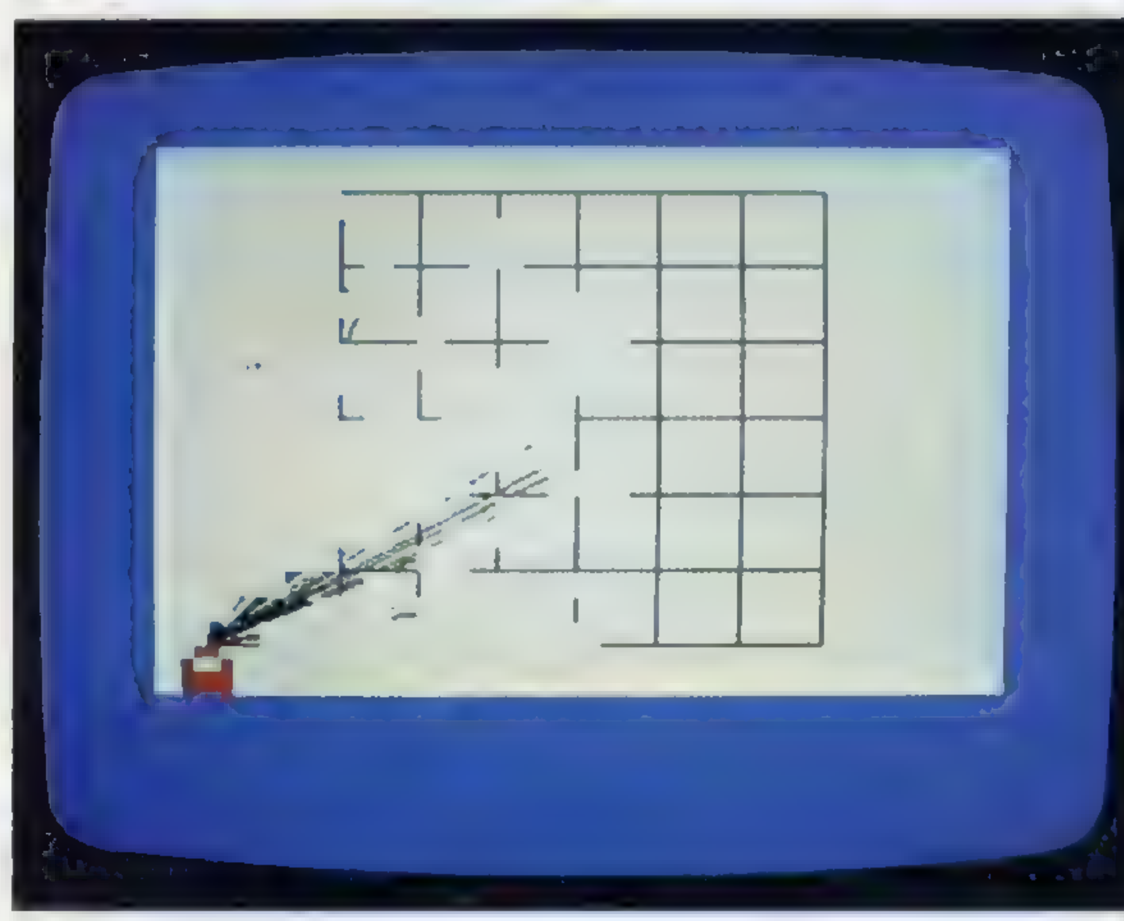
Lines 10 to 50 DRAW the grid – a black grid on a white background with a blue BORDER. Lines 60 and 70 PRINT a blue laser base in the bottom left corner of the screen. Line 80 PLOTs a point at the middle of the top of the laser base. This is simply a way of moving the graphics cursor to the top of the laser base, to the point where the laser beam will fire from. Line 90 produces values for x and y, which are the co-ordinates of the point to which the laser will fire – the end of the beam. The co-ordinates are set at random by the command RND (you will full details how to use this new on pages 48–49).

Line 100 DRAWs the black beam to x,y, while line 120 then "unDRAWs" the beam by DRAWing it in the background colour – white. Line 80 returns the program to the beginning of the firing routine again.

The next two screens show the display as it is when the program starts, and then as it is after the program has RUN for a while:

LASER AND GRID DISPLAYS





UnDRAWing and overPRINTing

You can see from the second display that the program doesn't do what is wanted. Firstly, when the beam is unDRAWn, the parts of the grid that lie along its path are also unDRAWn with it. Secondly, and perhaps more surprisingly, when each beam appears, so do the parts of all the previous beams that pass through character positions occupied by the current beam.

You could make the old beams invisible by adding INVERSE 1 to lines 100 and 110 like this:

```
100 DRAW INVERSE 1; INK 0;x,y
110 DRAW INVERSE 1; INK 0;-x,-y
```

The problem now is that, although the images of old

beams are eliminated, so is the current beam. Moreover, white squares still appear along the path of the beam, erasing parts of the black grid. You could try this instead:

```
100 DRAW INK 0;x,y
110 DRAW INVERSE 1;-x,-y
```

Now the beam will work properly, although its end point remains on the screen when the rest of the beam has been erased. The grid appears to be cut by white lines where the beam passes through it. The answer is to use OVER instead. Change lines 100 and 110 to:

```
100 DRAW OVER 1;INK 0;x,y
110 DRAW OVER 1;-x,-y
```

Now the beam works properly: it appears against the background of the grid and disappears again, without leaving any white cuts across the grid to mark its path. However, the end point of each beam remains on the screen. If this should fall on one of the grid lines, it leaves a white dot, breaking the line. This happens because even though the beam is DRAWn along one path and unDRAWn back down the same path back to its start point, the computer doesn't follow exactly the same path in both directions. So, to solve that, DRAW and unDRAW the beam along precisely the same path:

```
110 PLOT 16,16:DRAW OVER 1;x,y
```

Each beam is now DRAWn and unDRAWn from the same start point (16,16) to the same end point (x,y). The end points now no longer remain on the screen. The program works perfectly. The beams repeatedly flash across the grid and disappear again, leaving no evidence that they'd ever been there.

OverPRINTing with graphics

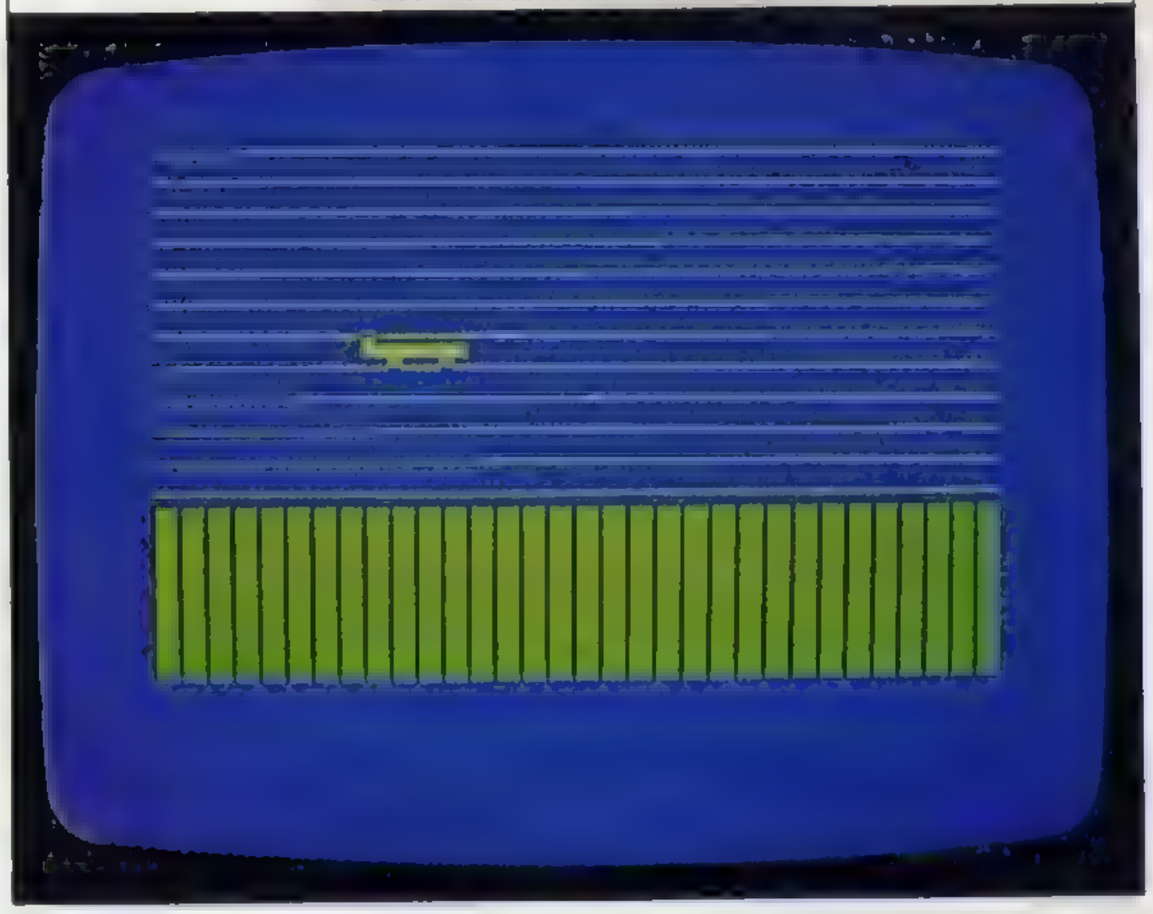
Now you can try replacing the laser beam with a larger graphics symbol, moving across a detailed background:

GRAPHICS WITH **OVER**

```
  10 BORDER 1: PAPER 1: CLS
  20 FOR r=15 TO 21
  30 FOR c=0 TO 31
  40 PRINT PAPER 4;AT r,c;" "
  50 NEXT c: NEXT r
  60 INK 0
  70 FOR x=0 TO 250 STEP 8
  80 PLOT x,55: DRAW 0,-55
  90 NEXT x
 100 INK 6
 110 FOR y=60 TO 170 STEP 10
 120 PLOT 0,y: DRAW 255,0
 130 NEXT y
 140 LET r=13: LET c=3
 150 PRINT OVER 1;AT r,c;"▙▄▄▄";
AT r+1,c+1;"▘ ▘"
 160 PAUSE 5
 170 PRINT OVER 1;AT r,c;"▙▄▄▄";
AT r+1,c+1;"▘ ▘"
 180 LET r=r-1: LET c=c+1
 190 IF r=0 THEN STOP
 200 GO TO 150

0 OK, 0:1
```

Lines 20 to 50 PRINT a green ground on a blue background, which forms a blue sky. Lines 60 to 90 DRAW a series of black perspective lines on the green ground to help give the impression of depth. Lines 100 to 130 DRAW a series of yellow lines across the sky. Line 110 controls their spacing as they are DRAWn upwards on the screen. Lines 140 to the end of the program PRINT a small aircraft on the ground and then make it take off and fly up and out of the top of the screen. Using OVER, the background is left untouched. The moving symbol has no effect on the lines previously DRAWn:

OVER GRAPHICS DISPLAY



To get some experience of using INVERSE and OVER, try changing the colours in these programs, and substituting PAPER colours for INK colours and vice versa. Testing these commands in short programs will soon give you ideas for using them in graphics.

When you do use INVERSE or OVER, remember that they are commands which must be activated or de-activated by the number that follows them. On their own they will not work.

How to BRIGHTen up your displays

As well as producing displays in seven different "real" colours, the Spectrum can produce displays of different colour intensities. The command used to change brightness is simply BRIGHT. This is used in PRINT statements in the same way as INVERSE and OVER. To turn on extra brightness, you use BRIGHT 1, and to turn it off, you use BRIGHT 0. If you type in these lines:

```
10 BORDER 0:PAPER 0:CLS
20 PRINT "without BRIGHT"
30 PRINT BRIGHT 1;"with BRIGHT"
```

you will be able to see the difference that this command makes. You can also use BRIGHT 8. This enables you to PRINT BRIGHT text without altering the original PAPER colour in the position to be PRINTed on.

SPECIAL SCREEN TECHNIQUES 2

Many programs rely for part of their effect on characters that flash on and off. With the Spectrum, you could make characters appear to flash by rapidly changing the INK and PAPER colours. But don't rush to your keyboard to try this, because there is a much better and simpler way to achieve flashing. The Spectrum has a single command, FLASH, which produces the effect you want.

How to turn flashing on and off

FLASH can have one of two numerical values, 0 or 1. FLASH 1 makes a character flash, while FLASH 0 stops it again. The next program will show you what effect adding FLASH has. First, type in the program without FLASH:

PROGRAM WITHOUT **FLASH**

Line 10 sets up the screen with a blue BORDER and PAPER, and white INK. The range of x values in line 20 (1 TO 704) is chosen because there are 704 character positions on the television screen (22 lines of 32 characters). Now to make the display flash, add a new instruction to line 10 to make it read:

```
10 FLASH 1:BORDER 1:PAPER 1:INK 7:CLS
```

When you RUN the program this time, the INK-PAPER colours alternate, making the display flash. A screen full of flashing characters is rather difficult to look at, and, after a short time, you will probably want to stop it. If you type:

```
FLASH 0
```

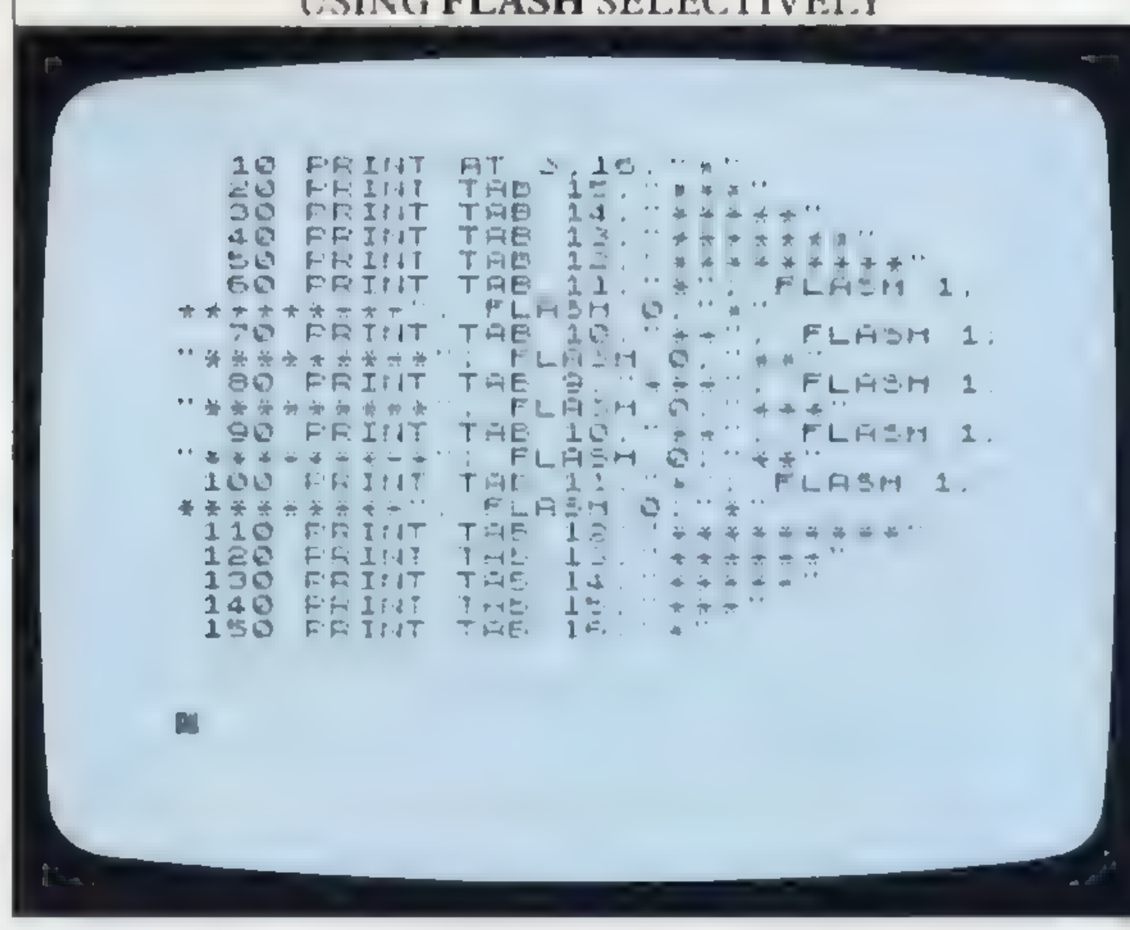
however, nothing happens. This is because the FLASH command only affects characters displayed after it. Instead, type CLS to remake the screen. The display should now stop flashing.

FLASH can be incorporated in PRINT statements in the same way as OVER and BRIGHT. All these commands can be used in one of two ways. If you now remove FLASH 1 from line 10 and key in:

```
30 PRINT FLASH 1;"#";
```

you will see the effect this has. Of course, you need not make the whole screen flash. In fact FLASH is a much more valuable command when used selectively. Try this display program:

USING **FLASH** SELECTIVELY



"Transparent" colour and contrast

Although the Spectrum has only eight screen colours, you may come across expressions using colour numbers 8 or 9. Colour number 8 produces a "transparent" colour. If you use PAPER 8 in a PRINT line, the PAPER colour will be left as it was before.

Colour number 9 does the exact opposite. If you've tried setting up your own colour screen, mixing the

available colours in different combinations, you will undoubtedly have discovered that some colours simply don't mix well. In some combinations, text disappears into the background, making it totally unreadable. INK 9 is very useful for achieving perfectly readable text first time. It PRINTs characters in either black or white, whichever contrasts more with the background PAPER colour:

INK 9 PROGRAM

```
  10 FOR p=0 TO 7
  20 INK 9: PAPER p: CLS
  30 PRINT AT 8,5;"INK 9 AUTOMAT
ICALLY"
  40 PRINT AT 10,5;"CONTRASTS IN
K COLOUR"
  50 PRINT AT 12,5;"WITH PAPER C
OLOUR"
  60 PAUSE 50
  70 NEXT p
```

Here a message is PRINTed against a background of each of the seven PAPER colours. The INK colour is chosen by the computer to contrast with the PAPER colour. White characters are PRINTed against black, blue, red and magenta backgrounds, but when the backgrounds change through green, cyan, yellow and white, the characters are PRINTed in black.

Overprinting without erasing

So far whenever you have PRINTed over text already on the screen, the original text has been blotted out by the new characters. However, this needn't always happen. The Spectrum keyword OVER which you met on the previous two pages works with text characters as well as graphics.

By using OVER, you can add characters together. For instance, lots of German words feature something called an *umlaut* – two dots above the letter a, o or u.

The Spectrum can generate accents, umlauts and double symbols with programs that overprint text like the first example in the three screens that follow. The same principle can be used to underline words on the screen, as you can see from the program in the centre screen in the next column. The underlining characters are PRINTed at exactly the same positions on the screen as one line of words. However, it doesn't, as you would normally expect, erase the words – it adds to them. You can see the result displayed on the bottom of the three screens. Try taking out line 20 and see the difference that it makes:

OVERPRINTING PROGRAMS

```
  10 PRINT AT 10,10;"Koln"
  20 PRINT OVER 1;AT 10,11;""""
  30 PRINT AT 12,10;"Societe"
  40 PRINT OVER 1;AT 12,14;"'"
  50 PRINT OVER 1;AT 12,16;"'"
  60 PRINT AT 14,10;"00000000"
  70 PRINT OVER 1;AT 14,10;"----
----"



0 OK, 0:1
```

```
  10 PAPER 0: BORDER 0: INK 7: C
LS
  20 OVER 1
  30 PRINT AT 8,8;"OVER LETS YOU
"
  40 PRINT AT 10,10;"UNDERLINE"
  50 PRINT AT 10,10;"_________"
  60 PRINT AT 12,6;"WORDS ON THE
 SCREEN"
```

By using OVER, you can convert letters into graphics characters, create foreign alphabets or, in games and graphics programs, you can make a character appear on top of a background line.

SOUND, NOTES AND MUSIC

The Spectrum can produce a wide range of sounds, all under the control of a single command, BEEP. This is accompanied by two variables – d (the duration or length of the sound) and p (pitch). For instance:

BEEP 1,0

produces a beep one second long at a pitch of middle C. The pitch of sounds is measured relative to middle C – above middle C, p is positive, below p it is negative. The Spectrum's pitch values range from −60 to +69. Increasing the value of p by 1 increases the pitch of the sound by one semitone. A semitone is the change in pitch between, for example, C and C#. Although d represents the length of the sound in seconds, it is not restricted to a whole number; fractions of seconds are quite permissible.

As you can see from the table of p values below, you can increase the pitch of a sound by an octave by adding 12 to p. You can play the full range of Spectrum notes by RUNning the following program. It takes about 15 seconds to complete the scale:

PITCH SCALE PROGRAM

```
  10 PRINT AT 10,5;"*************
********"
  20 PRINT AT 16,5;"*************
********"
  30 FOR p=-60 TO 69
  40 PRINT AT 12,9;"PITCH NUMBER
"
  50 PRINT AT 14,12;" ";p;" "
  60 BEEP 0.1,p
  70 PAUSE 10
  80 NEXT p



0 OK, 0.1
```

PITCH VALUES

Pitch values range from −60 to 69. The values for six octaves grouped around middle C (0) are shown here.

Note	Pitch Value					
G#, Ab	−16	−4	8	20	32	44
G	−17	−5	7	19	31	43
F#, Gb	−18	−6	6	18	30	42
F	−19	−7	5	17	29	41
E	−20	−8	4	16	28	40
D#, Eb	−21	−9	3	15	27	39
D	−22	−10	2	14	26	38
C#, Db	−23	−11	1	13	25	37
C	−24	−12	0	12	24	36
B	−25	−13	−1	11	23	35
A#, Bb	−26	−14	−2	10	22	34
A	−27	−15	−3	9	21	33

A FOR ... NEXT loop in lines 30 to 80 goes through all the possible p values from −60 to 69, sounding each note for a tenth of a second, and lines 10 and 20 PRINT a frame around the pitch number PRINTed by line 50.

Measuring the Spectrum's speed with sound

The next program uses BEEP as a signal. If you want to find out how quickly your Spectrum works, you can write a program to make it carry out a long series of calculations, and then time how long it takes to complete the series. For an approximate timing, your watch will be accurate enough, and to get around the problem of having to look at your watch and the screen at the same time, you can use BEEP to mark the beginning and end of the program. Here is one way by which you can do it:

SPEED TEST PROGRAM

```
  10 PRINT AT 6,4;"SPECTRUM SPEE
D TEST"
  20 PAUSE 100
  30 LET t=0  BEEP 0.1,25
  40 PRINT AT 9,4;"calculation s
ubtotal"
  50 FOR c=1 TO 1000
  60 LET t=t+c
  70 PRINT AT 12,11;t
  80 NEXT c
  90 BEEP 0.1,30
 100 PRINT AT 9,4;"     FINAL TOT
AL     "



0 OK, 0 1
```

```
     SPECTRUM SPEED TEST

         FINAL TOTAL

            500500



0 OK, 100 1
```

After the program title has appeared, lingered a while (determined by PAUSE 100), the timing period begins.

A message on the screen and a BEEP lasting 1/10th of a second tell you when the calculation – adding together all the numbers from 1 to 1000 – has started. Some seconds later, there is a second BEEP, again lasting 1/10th of a second but at a higher pitch, marking the end of the timing period. In that time, your Spectrum has not only performed 1000 calculations, but it has also PRINTed out 1000 results! Dividing the total time by 1000 will tell you how long each calculation and PRINTing takes.

Programming simple tunes

With the BEEP command, it is quite easy to get the Spectrum to play a simple tune. Here to start with is a short sequence of notes. You just enter a BEEP command for each separate note:

SIMPLE NOTE SEQUENCE

```
10 BEEP 0.5,0
20 BEEP 0.5,4
30 BEEP 0.5,7
40 BEEP 0.5,12
50 BEEP 0.5,7
60 BEEP 0.5,4
70 BEEP 0.5,0

0 OK, 0:1
```

As all the BEEPs are 0.5 seconds long, you could save a little typing time by using a variable, t, and setting it to 0.5 at the beginning of the program. The BEEP statements would then look like this:

NOTE SEQUENCE – VARIABLE DURATION

```
 5 LET t=0.5
10 BEEP t,0
20 BEEP t,4
30 BEEP t,7
40 BEEP t,12
50 BEEP t,7
60 BEEP t,4
70 BEEP t,0

0 OK, 0:1
```

The advantage of this is that you can then change the value of t at the beginning of the program, and all the timings will then alter in step.

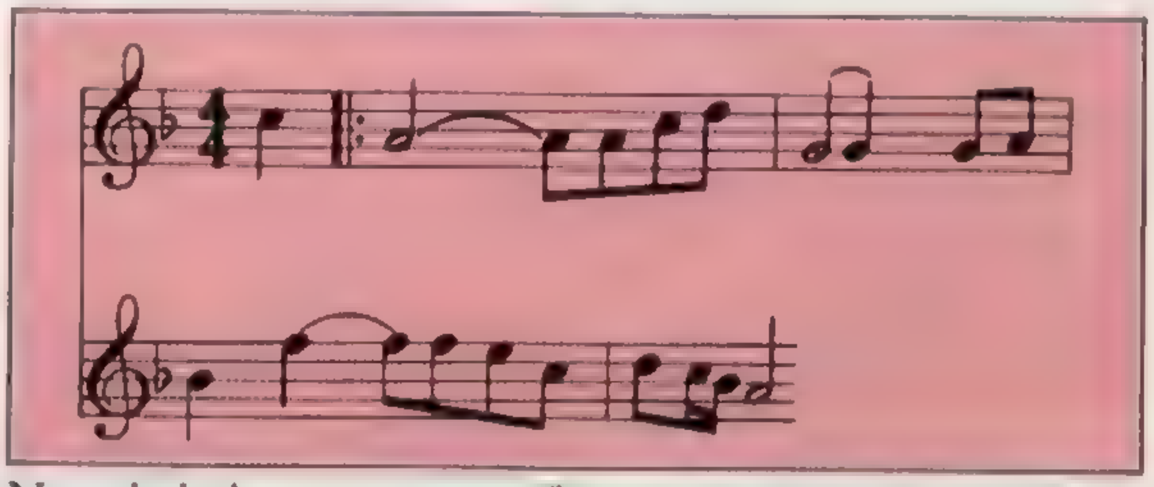
Music on the Spectrum

If you want to write real music on the Spectrum, you will have to tackle the problem of timing. Musical notes can be of various lengths. If you call the duration of a crotchet d, then the durations of the other notes are related to it like this:

NOTE DURATIONS

If the duration of a crotchet is set at a variable d, the durations of other notes are as below.

Semi-quaver (sixteenth note)	d/4
Quaver (eighth note)	d/2
Crotchet (quarter note)	d
Minim (half note)	2*d
Semi-breve (whole note)	4*d

You can then set about converting a piece of sheet music into a computer program. Here are a few bars of a tune which you will probably recognize when you get your Spectrum to play them. The musical score for the first few bars looks like this:

Now it is just a matter of converting the notes. This listing sets d at 0.5; once you have entered the program you can vary the speed by altering this value. You will find that the pitch remains unaltered:

TUNE PROGRAM

```
  5 LET d=0.5
 10 BEEP d,12
 20 BEEP 2.5*d,9
 30 BEEP d/2,9
 40 BEEP d/2,12
 50 BEEP d/2,14
 60 BEEP 3*d,7
 70 BEEP d/2,7
 80 BEEP d/2,9
 90 BEEP d,10
100 BEEP 1.5*d,17
110 BEEP d/2,17
120 BEEP d/2,16
130 BEEP d/2,12
140 BEEP d/2,14
150 BEEP d/4,12
160 BEEP d/4,10
170 BEEP 2*d,9

0 OK, 0:1
```

SPECIAL EFFECTS WITH SOUND

So far you have just used the BEEP command to produce musical notes. However, when you come to writing your own programs, you will often want some quite unmusical sound effects to give the screen display added realism. Menacing sounds add a whole new dimension to many programs.

Writing sound loops

The simplest sound effect you can produce uses two BEEPs, at different pitches, and then cycles between the two by using GOTO to make a loop. Here is a program that produces a siren effect:

GOTO SOUND LOOP

This simply plays a high note for a quarter of a second, followed by a lower note of the same length. You could write both BEEP statements on the same line so that the whole sound routine would take up only two lines:

```
10 BEEP 0.25,9:BEEP 0.25,7
20 GOTO 10
```

Because this program is written as an endless loop, you will need to press the CAPS SHIFT and BREAK keys to make it stop. However, by changing the loop to use FOR … NEXT, you could incorporate it into a program as a sound effect that lasts for a set period of time, or that is repeated at intervals.

Altering the duration of a sound loop

The character of a sound can be changed dramatically if you shorten the playing time of the elements that make it up. By altering t in line 10 of either of the next two BEEP programs, you can hear the effect of shortening a sound sequence. These sorts of sound are often used in otherwise routine games to startle the player at a critical moment. The shorter the duration of the BEEPs, the more urgent the sound seems to become:

LOOP SOUND EFFECTS

The sound programmed in the third screen illustrates what happens when the BEEP gets really short. The program uses just one BEEP statement, but it features a technique that you haven't come across before – that of stepping backwards through a loop. The loop begins at p=60 and decreases p by 2 on each cycle until p=30. Each note sounds for only 0.005 (5 thousandths) of a second. If you make the note any shorter than this, it will sound like a click. You cannot produce a great variety of sounds with such simple sound commands. However, if you use sounds at the bottom end of the pitch scale, you can produce another type of effect. Try this program; it produces a chugging sound, rather like an engine:

LOW PITCH LOOP

Unpredictable sounds

So far, you have had a good idea of what sound a program would have produced before you ran it. But now try this program. (You will find the keyword RND above the T key):

RANDOM SOUND

Instead of giving the pitch a fixed value or a predictable range of values, this program lets the computer choose pitches at random. (The use of RND is covered fully on pages 48–49).

Synchronized sound effects

It's relatively easy to generate sounds in isolation, as you have been doing so far, but incorporating sound effects in a program successfully is more difficult. The trick is to get the sound at the right duration at the right part of the program. To illustrate how this is done, the next program makes a simple graphics character move from one side of the screen to the other, while generating a sound effect in step with the movement:

SYNCHRONIZED SOUND AND ANIMATION

Instead of setting up user-defined graphics, the program uses the relatively coarse graphics available on the number keys. The character is composed of a total of eight separate symbols PRINTed over three lines. Normally, the character would be PRINTed and then, after a short time delay, erased before being PRINTed again one position further across the screen.

The delay makes sure that the character is on the screen longer than it is off, to minimize flickering. Here, though, the sound effect itself is used as a time delay. If one sound were made on each pass round the FOR … NEXT loop (lines 20 to 130), it would be a rather disjointed, stuttering sound because of the relatively long silences. Splitting the sound effect into a number of different BEEP statements gets round that. It also makes more complicated sound effects possible than a single BEEP statement could produce. The lengths of the sounds are carefully chosen so that there is as little delay as possible between one character being erased and the next character being PRINTed. Since the BEEP command stops the program for as long as the sound is generated, it is necessary to display the character on the screen first and to generate the sound immediately afterwards.

DECISION-POINT PROGRAMMING

You have already looked at the concept of the loop in programs a number of times from page 26 onwards. If you want to carry out a calculation or put something on the screen 10 times, you could write:

FOR A=1 TO 10 ...
NEXT A

But there is another way of doing this, by using an IF ... THEN statement. To take an example, let's say you want to PRINT all the numbers from 1 to 10, together with their squares and cubes in a table. Here is how you would do it with FOR ... NEXT, and then with a different program which uses IF ... THEN:

FOR...NEXT LOOP

```
  10 BORDER 1: PAPER 6: INK 2: C
LS
  20 PRINT AT 4,6;"A";AT 4,14;"A
↑2";AT 4,24;"A↑3";AT 5,0;"________________________________"
  30 FOR n=1 TO 10
  40 PRINT
  50 BEEP 0.1,40
  60 PRINT AT n+6,6;n;AT n+6,14;
n↑2;AT n+6,24;n↑3
  70 NEXT n

0 OK, 0:1
```

In the IF ... THEN program which follows, line 10 sets up the colour screen, and line 30 PRINTs the table's heading as before. Line 40 is the first line of the loop – it increases n by 1 on every pass round the loop. Line 70 is the same PRINT statement used in the FOR ... NEXT program. Line 80 is where the computer makes a decision as it examines n. The < symbol is mathematical shorthand for "less than". So, if n is less than 10, the computer is told to go around the program again from line 40. The computer continually PRINTs out A, A ↑ 2 and A ↑ 3 until n is not less than 10 when, in this case, it stops:

IF...THEN LOOP

```
  10 BORDER 1: PAPER 6: INK 2: C
LS
  20 LET n=0
  30 PRINT AT 4,6;"A",AT 4,14;"A
↑2";AT 4,24;"A↑3";AT 5,0;"________________________________"
  40 LET n=n+1
  50 PRINT
  60 BEEP 0.1,40
  70 PRINT AT n+6,6;n;AT n+6,14;
n↑2;AT n+6,24;n↑3
  80 IF n<10 THEN GO TO 40

0 OK, 0.1
```

Why use the IF ... THEN loop?

You might wonder what the point of this is, as the IF ... THEN loop produces just the same results as the FOR ... NEXT loop. The value of IF ... THEN is that the computer can respond to any information that you INPUT during the program's operation by making a decision about it. Here is an example which shows this, by giving you a chance to test your skill at mental arithmetic (RND is explained on pages 48–49):

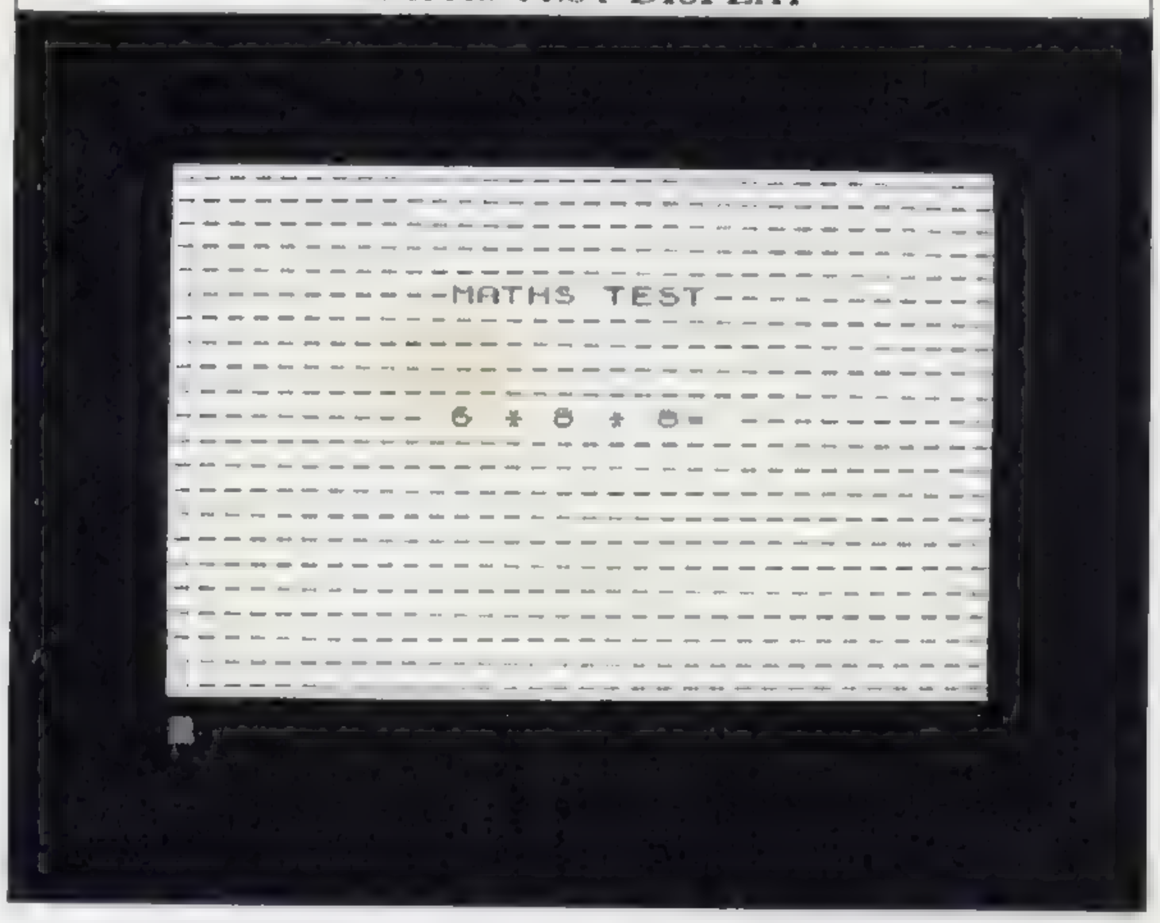
MATHS TEST PROGRAM

```
  10 BORDER 0: CLS
  20 FOR r=0 TO 21
  30 FOR c=0 TO 31
  40 PRINT AT r,c;"-"
  50 NEXT c: NEXT r
  60 PRINT AT 5,11;"MATHS TEST"
  70 LET x=INT (RND*10): LET y=I
NT (RND*10): LET z=INT (RND*10)
  80 PRINT AT 10,10;" ";x;" * ";
y;" * ";z;"= ": INPUT a
  90 IF a=x*y*z THEN GO TO 130
 100 BEEP 0.5,40: PRINT AT 10,10
;"---Wrong------": PAUSE 50
 110 PRINT AT 10,10;"-Try again-
": PAUSE 50
 120 GO TO 80
 130 BEEP 0.5,50: PRINT AT 10,10
;"--Correct---": PAUSE 50
 140 GO TO 70

0 OK, 0.1
```

Each time the computer sets the problem and waits for your answer, it is faced with two possible courses of action. If you type in a correct answer, the IF ... THEN statement at line 90 directs the computer to go, not to line 100, but to line 130 next – PRINTing a "correct" message and then setting another problem. If the answer is wrong, then the computer "falls through" the IF ... THEN statement to line 100 and goes into the "wrong" routine.

It is important to remember that there must also be something in the program to stop the wrong answer routine carrying on into the correct answer routine. In this case, it is line 120, which makes the computer PRINT out the problem again:

MATHS TEST DISPLAY



Creating graphics with IF ... THEN

You can use IF ... THEN in combination with graphics commands to turn your Spectrum into an electronic drawing system. All you have to do is program the computer to respond to an INPUT by DRAWing:

IF...THEN GRAPHICS PROGRAM

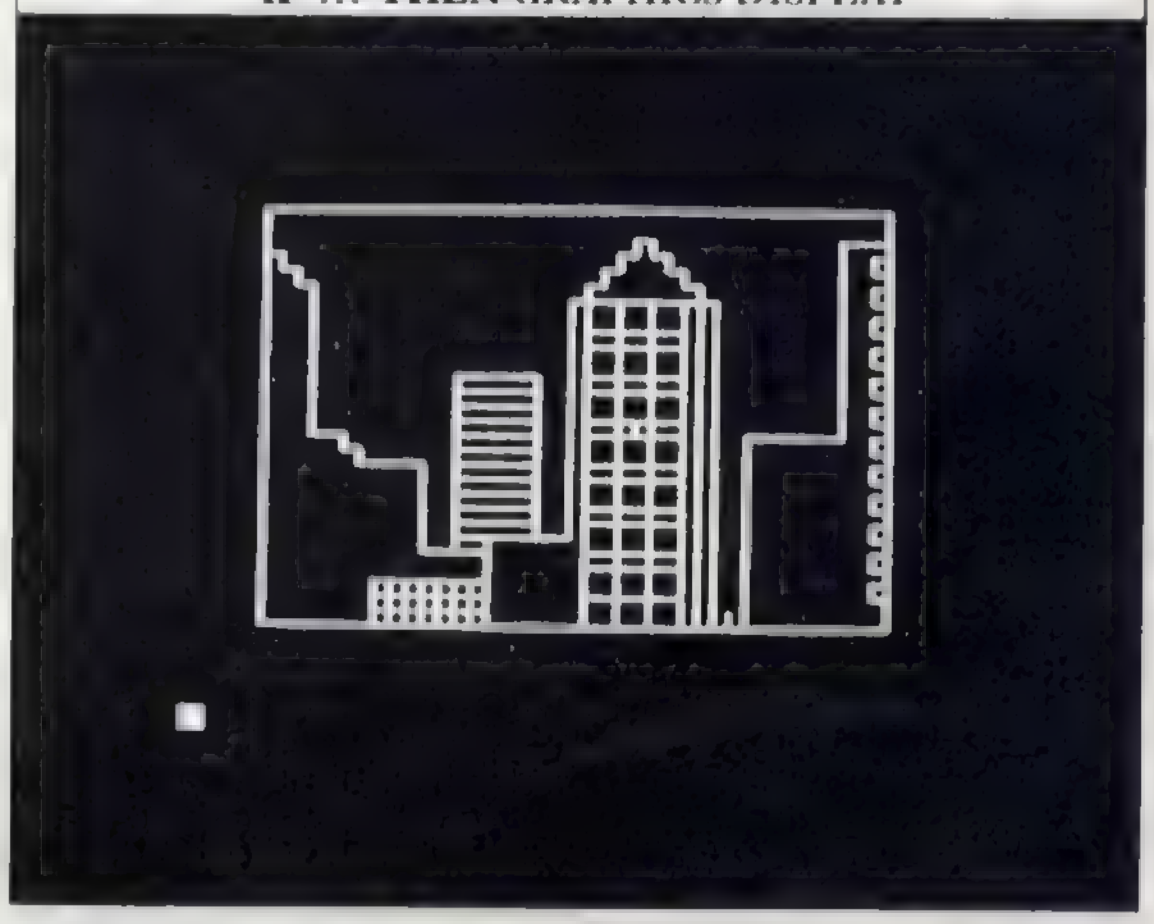
```
  10 BORDER 0: PAPER 0: INK 7: C
LS
  20 PLOT 25,25
  30 LET x=5
  40 INPUT k
  50 IF k=5 THEN DRAW -x,0
  60 IF k=6 THEN DRAW 0,-x
  70 IF k=7 THEN DRAW 0,x
  80 IF k=8 THEN DRAW x,0
  90 GO TO 40

0 OK, 0:1
```

In this simple program the IF ... THEN statements allow the computer to decide what action to take. This program uses the cursor keys to DRAW lines either horizontally or vertically from a point near the bottom left of the screen.

The program simply PLOTs the point 25,25 and then DRAWs a small line every time you press one of the cursor keys. The four IF ... THEN lines make the computer examine your INPUT, and then decide in which direction the line should be DRAWn. Each time you use a key, remember to press ENTER so that the computer receives the instruction.

The statement in line 30 tells the computer how long to make each line. Making x equal to 5 gives quite good results, but you might like to try altering its value to change the resolution of your pictures. In addition to this, of course, you can change colour simply by altering the numbers in line 10. This screen will give you an idea of what you can DRAW:

IF ... THEN GRAPHICS DISPLAY



Selecting the right condition

Although you can extend this sort of program to use more keys, the Spectrum does have a better way of achieving the same effect, and having long rows of IF ... THEN statements is not really good programming. But when you use IF ... THEN, remember that there is a great variety of "conditions" which can follow the IF part of the statement. The programs on these pages have used either < or =, but this is only part of the complete range of symbols that the Spectrum uses as you can see from the following table. Choosing the right condition is not always easy, especially when you are dealing with moving characters.

IF ... THEN CONDITIONS

The symbols that follow the IF part of a line specify the kind of decision that the computer will make.

= is equal to	<> is not equal to
> is greater than	< is less than
>= is greater than or equal to	<= is less than or equal to

UNPREDICTABLE PROGRAMS

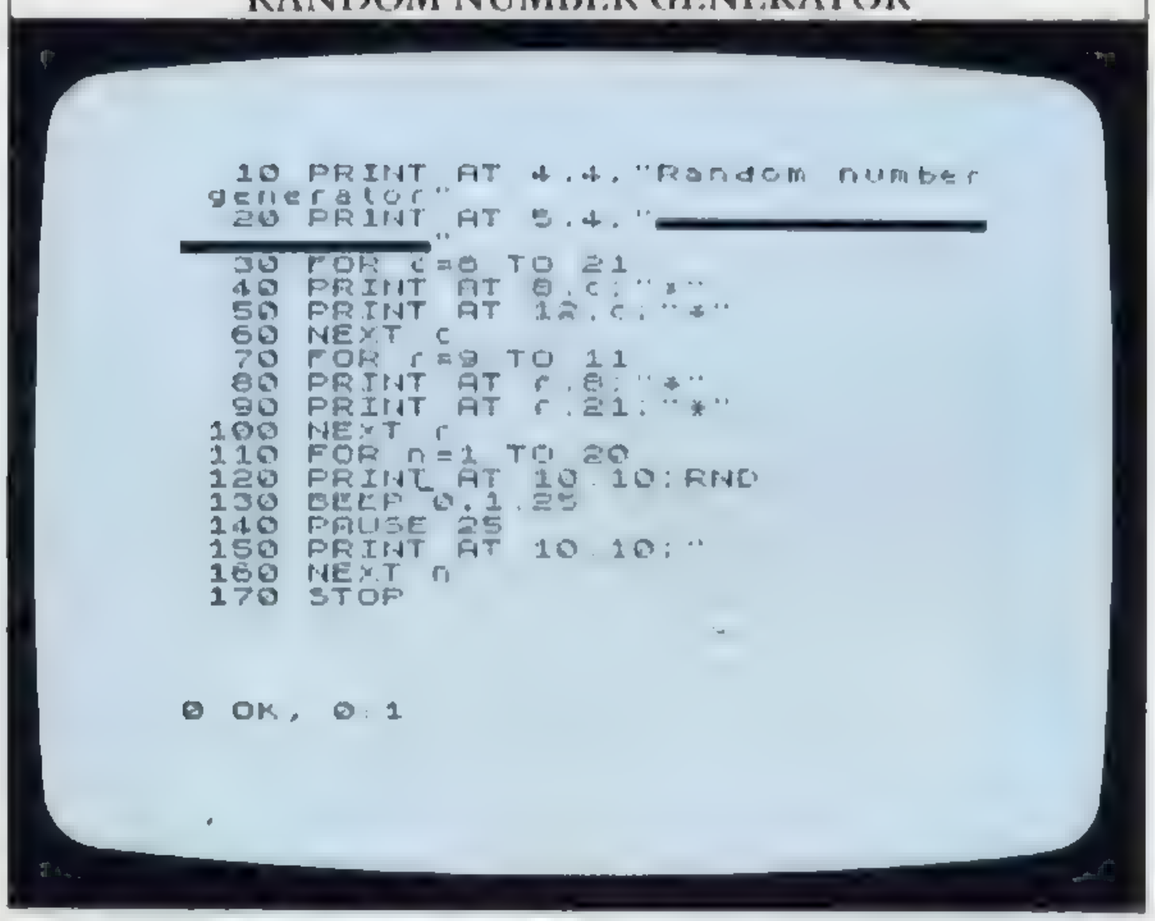
Although computers generally work with precise information, doing exactly what you tell them to do, an element of chance is necessary in certain applications. For instance, most computer games are based to some extent on luck. If you want to make something happen at an unpredictable time, or if dice are to be thrown or coins tossed, you can't tell the computer what result to produce every time or the element of chance would disappear.

The way to build chance into a program is to use RND. You will already have come across this command – it was used to produce a series of random numbers for example in the maths test program on the previous two pages. RND, as you have probably guessed, stands for RaNDom and it allows you to generate random numbers up to a maximum that you can set. You can then use these numbers to produce unpredictable sequences. The command is used like this:

```
10 A=RND
```

This will make A a decimal number that is somewhere between 0 and 0.999999999. Try using RND in this program, which PRINTs numbers at random:

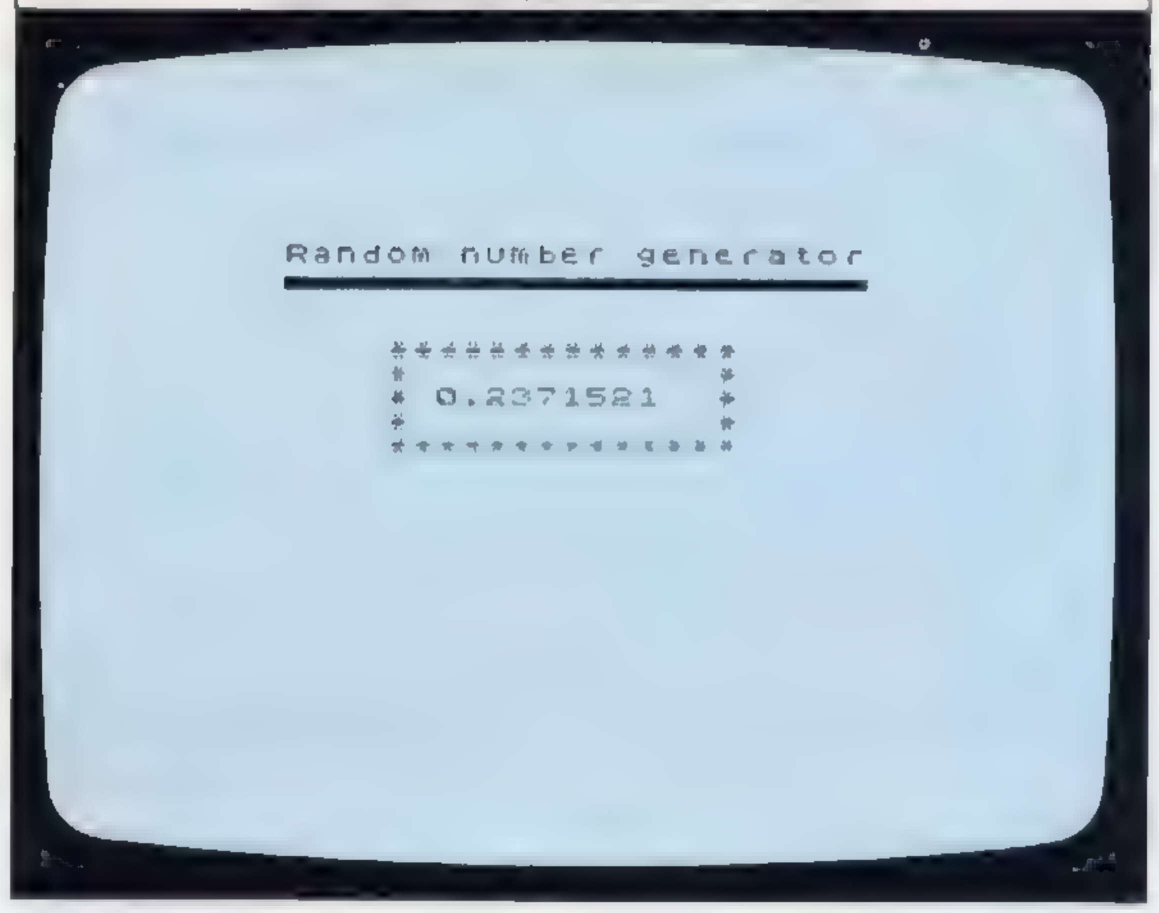
RANDOM NUMBER GENERATOR



This uses RND in line 120 to generate random numbers between 0 and 0.999999999, while lines 30 to 100 set up a border of asterisks to frame the numbers. Very small numbers include the E symbol that you came across on page 17. Normally, as each new number is PRINTed, it automatically erases the last number – simply by PRINTing on top of it. However, when something like E-4 appears, it is not automatically erased, so the nine blank spaces in line 150 take care of that.

The Spectrum can generate only the limited range of random numbers used in this program. To generate other random numbers, you have to manipulate RND.

RANDOM NUMBER DISPLAY



Producing random whole numbers

If you now replace line 120 with:

```
120 PRINT AT 10,14;INT (RND*10)
```

and RUN the program again, you will notice an immediate change in the display. The numbers are no longer decimal fractions, in fact there's no decimal point at all. Instead, the program is generating whole numbers between 0 and 9 inclusive – a much more useful result for programs. INT, which you came across on page 27, rounds the random number produced down to the nearest whole number, or integer.

This way of using RND is very useful for programming games with chance built in. It is quite easy, for example, to get the Spectrum to simulate throwing dice or tossing coins:

COIN TOSS PROGRAM

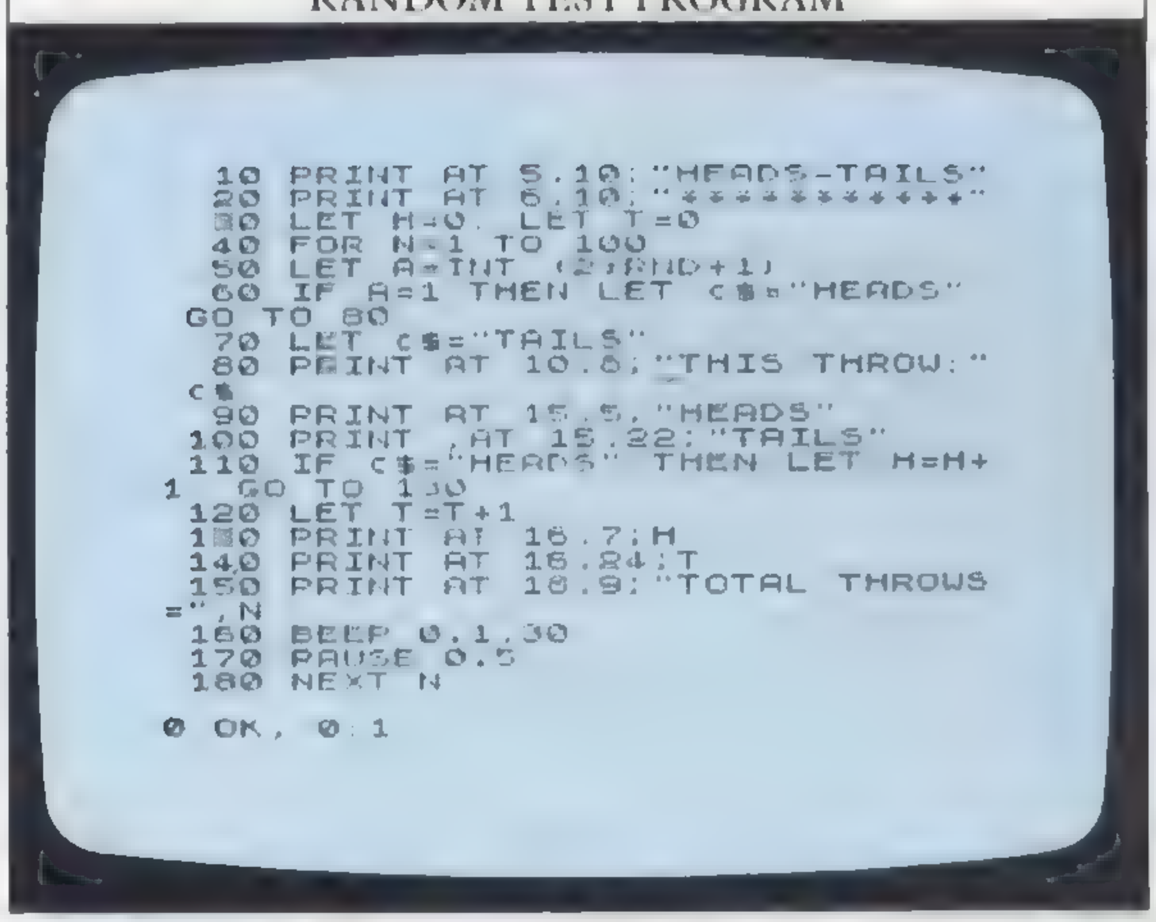
As a tossed coin can only have one of two values – heads or tails – line 20 produces a random number that is either 1 or 2. Tails are represented by 1 and heads by 2. Two IF ... THEN lines determine what is to be PRINTed, and then the program continues:

COIN TOSS DISPLAY

Checking a random sequence

It is possible to write a program that will show you just how random RND is. If you use RND to toss an electronic "coin" 100 times, you should get roughly 50 heads and 50 tails each RUN. You can actually test to see if this is true. Key in this program:

RANDOM TEST PROGRAM



When you RUN this, you will be able to see how close to 50:50 the heads and tails are each time the program is carried out.

Using RND in graphics programs

You can produce some interesting effects with RND by incorporating it in graphics programs, so that the computer is instructed to PRINT a character at a random position on the screen. If you then make the computer repeat this a number of times by using FOR...NEXT you can build up a display which will be different every time the program is RUN. Here is a program which uses RND in this way:

RANDOM GRAPHICS PROGRAM

Lines 10 to 80 define a character that is stored in the computer's memory and recalled by using the A key with the graphics cursor. Line 90 sets up a screen which is completely black – both BORDER and PAPER are set to 0. Lines 100 to 150 make up a FOR ... NEXT loop which selects a random position and a random INK colour, and which then PRINTs the character. PAUSE 0 (line 160) simply stops the program until any key is pressed, preventing the completion report from appearing:

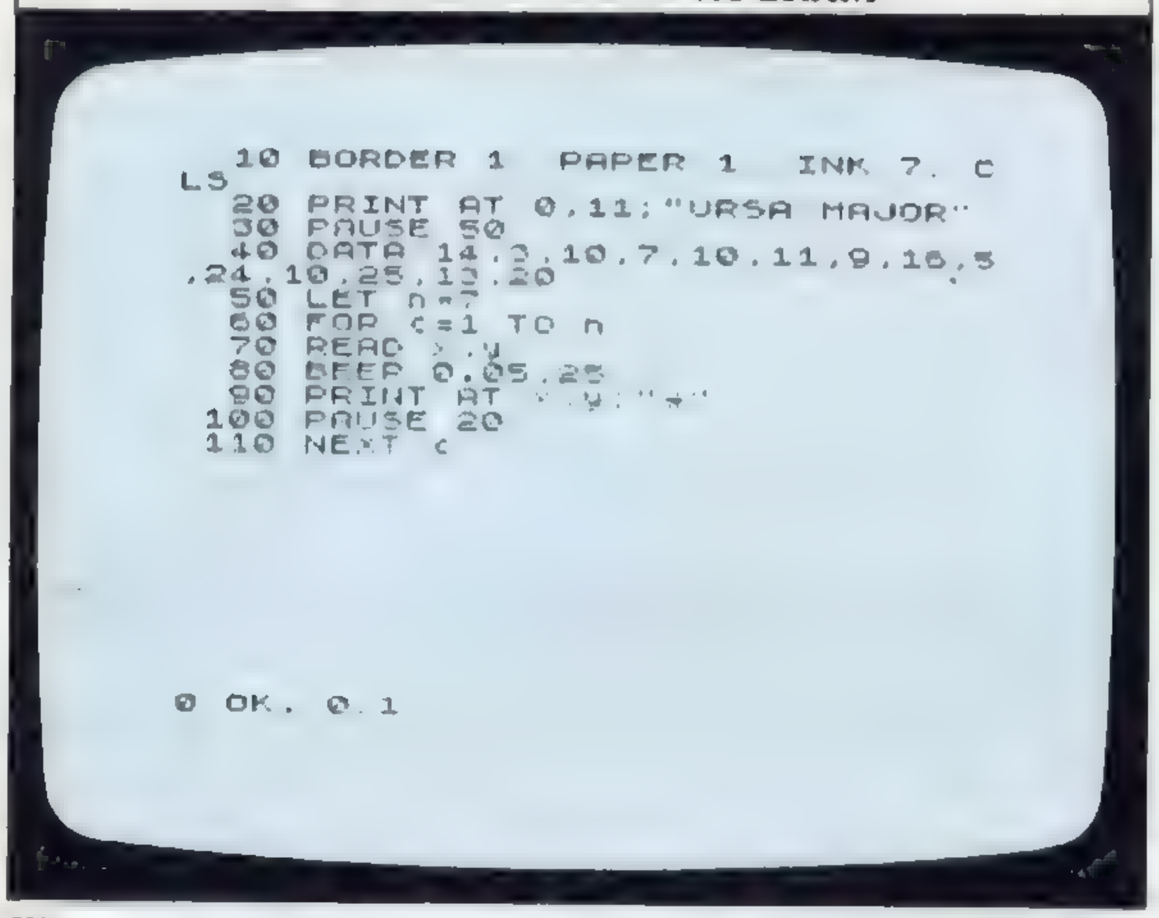
RANDOM GRAPHICS DISPLAY

You can also use the RND command in programs to make keyboard characters, or even random points, appear in random colours. To do this, use the command INK c (where c is a random number between 1 and 7) in the same way as in line 140 of the program above.

COMPILING A DATA BANK

The data necessary for a program can be collected while it is RUNning by using INPUT, or alternatively can be written into the program itself. The commands used to store data are quite straightforward. Data is held in DATA statements and read by READ statements. Here is a program which will show you the technique at work. Type in:

CONSTELLATION PROGRAM



When you RUN this program you should see on your screen a computer-generated map of a group of stars, the constellation Ursa Major, also known as the Plough or Big Dipper:

CONSTELLATION DISPLAY

The information for the display is carried in line 40 in the form of 14 co-ordinates. Line 70 tells the computer to READ the DATA in line 40, and to understand the DATA as pairs of figures which the program will refer to as x,y. Line 50 tells the computer that there will be seven of these pairs altogether.

The computer first produces a short pulse of sound, and then PRINTs an asterisk AT each value of x,y, transforming the row of DATA into a map on the television screen.

With a program like this it is easy to enter new DATA to get the computer to PRINT a new map. Here is a set of line changes and the map it produces:

```
20 PRINT AT 0,11;"CASSIOPEIA"
40 DATA 8,3,12,8,9,14,14,18,8,24
50 LET n=5
```

CONSTELLATION DISPLAY

When you use DATA statements, it is important to tell the computer how much DATA there is to READ. Line 50 in the constellations program shows you how to do this. It sets the limit for the number of pairs of co-ordinates that are to be READ, so when the computer has PRINTed the final star, it stops. If there was no FOR ... NEXT loop in the second half of the program, the computer would run out of DATA. If this happened the program would end with an error report.

Storing numbers and strings together

Strings, too, can be stored and READ using DATA lines, and you can also store a mixture of both numbers and strings – the names of friends and their phone numbers or birthdays, for example. This does present a problem though, because two different types of READ statement are used to read numbers and strings – READ a and READ a$, for instance. But if you make all the DATA, including the numbers, appear as strings, you can overcome this difficulty.

The following program holds a personal telephone list. Names and telephone numbers are loaded by lines 10 to 50. Lines 60 to 80 display the program title and then offer a choice of functions:

TELEPHONE LIST PROGRAM

```
  10 DATA "C.Barnes","141","R.Bu
ckman","322","J.Hann","166"
  20 DATA "R.Harty","103","S.Ing
le","191","S.Jay","47"
  30 DATA "H.Kelly","33","M.Park
inson","86","M.Philbin","71"
  40 DATA "B.Redhead","122","M.R
odd","100","S.Scott","260"
  50 DATA "M.Stoppard","300","J.
Timpson","90"
  60 PRINT AT 5,4;"PERSONAL TELE
PHONE LIST"
  70 PAUSE 100
  80 PRINT AT 12,2;"COMPLETE LIS
TING....press 1";AT 15,2;"SELECT
IVE LISTING...press 2": INPUT c
  90 CLS
 100 IF c=2 THEN GO TO 160
 110 PRINT : PRINT : PRINT
 120 FOR c=1 TO 14
 130 READ n$,m$
 140 PRINT TAB 6;n$;TAB 18;m$

scroll?
```

```
 150 NEXT c: PAUSE 0: CLS : GO T
O 60
 160 CLS : PRINT AT 9,5;"ENTER I
NITIAL AND NAME"
 170 INPUT e$
 180 LET r$="Name not found"
 190 RESTORE
 200 FOR c=1 TO 14
 210 READ n$,m$
 220 IF n$=e$ THEN LET r$=n$+"
.......   "+m$
 230 NEXT c
 240 CLS : PRINT AT 10,5;r$
 250 PAUSE 200: CLS
 260 RESTORE : GO TO 60




0 OK, 0:1
```

To PRINT the whole telephone list, type:

1 then ENTER

COMPLETE TELEPHONE LIST

```
C.Barnes     141
R.Buckman    322
J.Hann       166
R.Harty      103
S.Ingle      191
S.Jay        47
H.Kelly      33
M.Parkinson  86
M.Philbin    71
B.Redhead    122
M.Rodd       100
S.Scott      260
M.Stoppard   300
J.Timpson    90
```

Alternatively, by typing in the following command:

2 then ENTER

you can find the number of just one name, after which the computer returns to the selection display:

TELEPHONE LIST SELECTION DISPLAY

```
PERSONAL TELEPHONE LIST


COMPLETE LISTING....press 1

SELECTIVE LISTING...press 2
```

If you type in 2 at line 80, the program follows lines 160 to 260. You are first asked to enter an initial and a name. Make sure that you PRINT in capitals and lower case as in the DATA lines, without spaces, or the computer will not recognize your entry.

If the computer finds that the name (e$) that you typed in is the same as one of the names (n$) in the DATA statements, it will give a new string (r$) the value of n$ plus a line of dots and the telephone number. If it does not find e$, r$ is left unchanged at "Name not found" (set by line 180), and that is PRINTed out at the end of the program.

Because you want to add the name, a line of dots and the telephone number together in line 220, the telephone number has to be treated as a string variable m$, instead of a numeric variable, m. If you used m the program would not work because string and numeric variables cannot be added together.

Lines 190 and 260 use a new command, RESTORE. This tells the computer to go back to the beginning of the DATA statements when it carries out the next READ command. Without it, the program would only RUN correctly once. This would be because the computer would run out of DATA; it would try to continue searching the DATA after the final item, but the program tells it to READ all the available DATA on each RUN. RESTORE lets the computer start searching from the beginning of the DATA each time as if it was the first RUN.

Once you know how to compile a DATA bank, you can use one to store your friends' birthdays, another to list bills and payments, or the details of your videotape or cassette library.

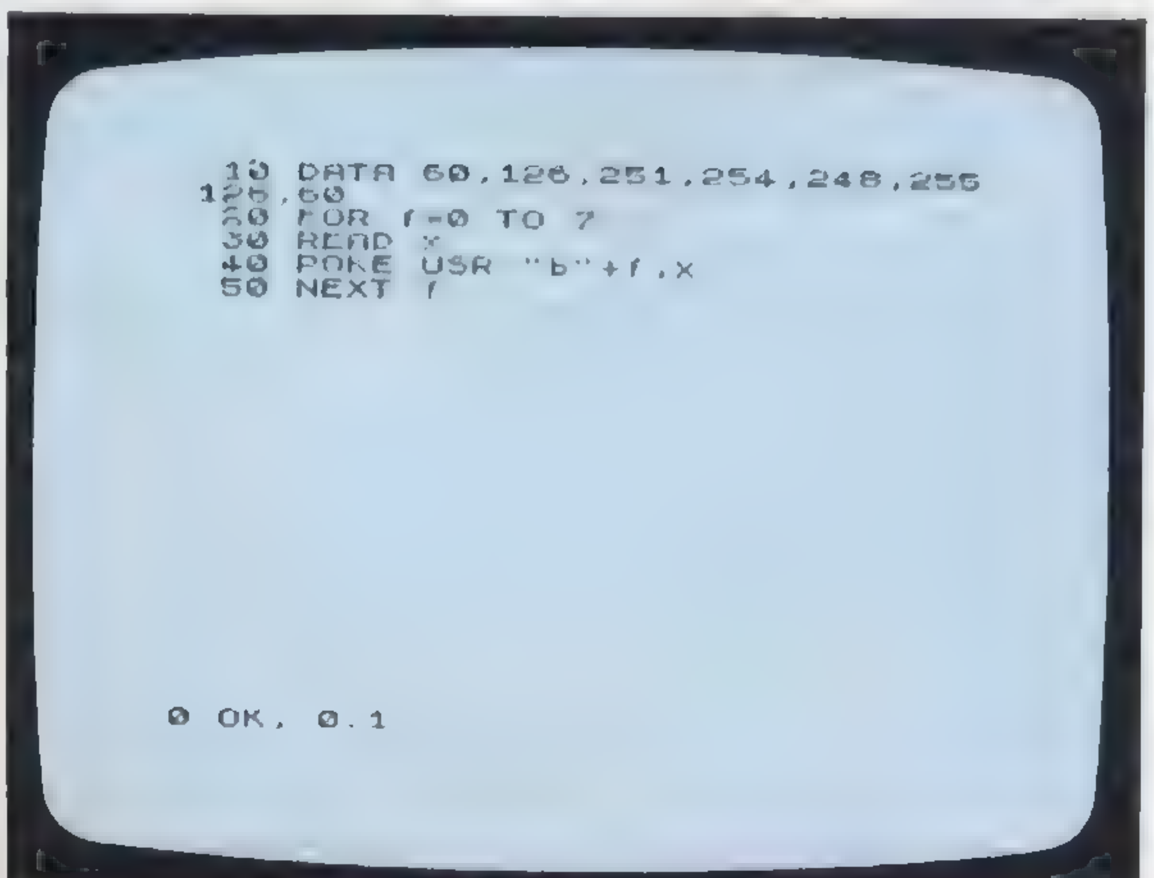
QUICK WAYS TO STORE CHARACTERS

On pages 30–31 you saw how to program and store your own graphics characters using the commands POKE and USR. As you will have noticed, getting the computer to produce even one user-defined character is quite a lengthy business. Programming a single graphics key takes eight program lines, so if you want to produce a symbol made up from four separate characters, you would need 24 program lines. However, the graphics programs on pages 36–37 used a short-cut that can make producing your own characters much faster.

Programming characters with READ ... DATA

As you saw on page 30, to program a graphics character you need to enter the numeric value of each line in the character grid. Here are two programs which produce the same character. The first uses a separate program line to store each separate total from the character grid:

SAVING LINES WITH **FOR...NEXT**



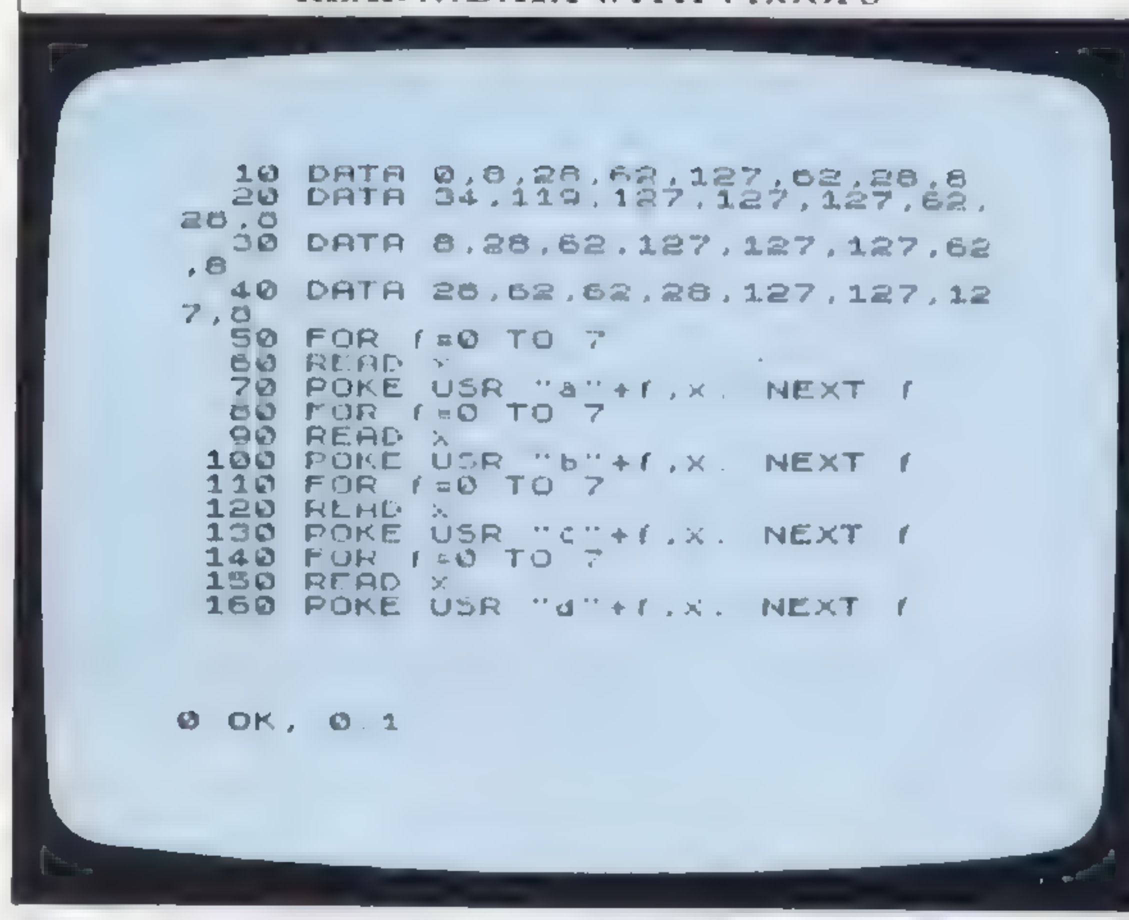
The second program instead stores the totals in a DATA line at the beginning of the program, and then converts them into the same character using a READ line.

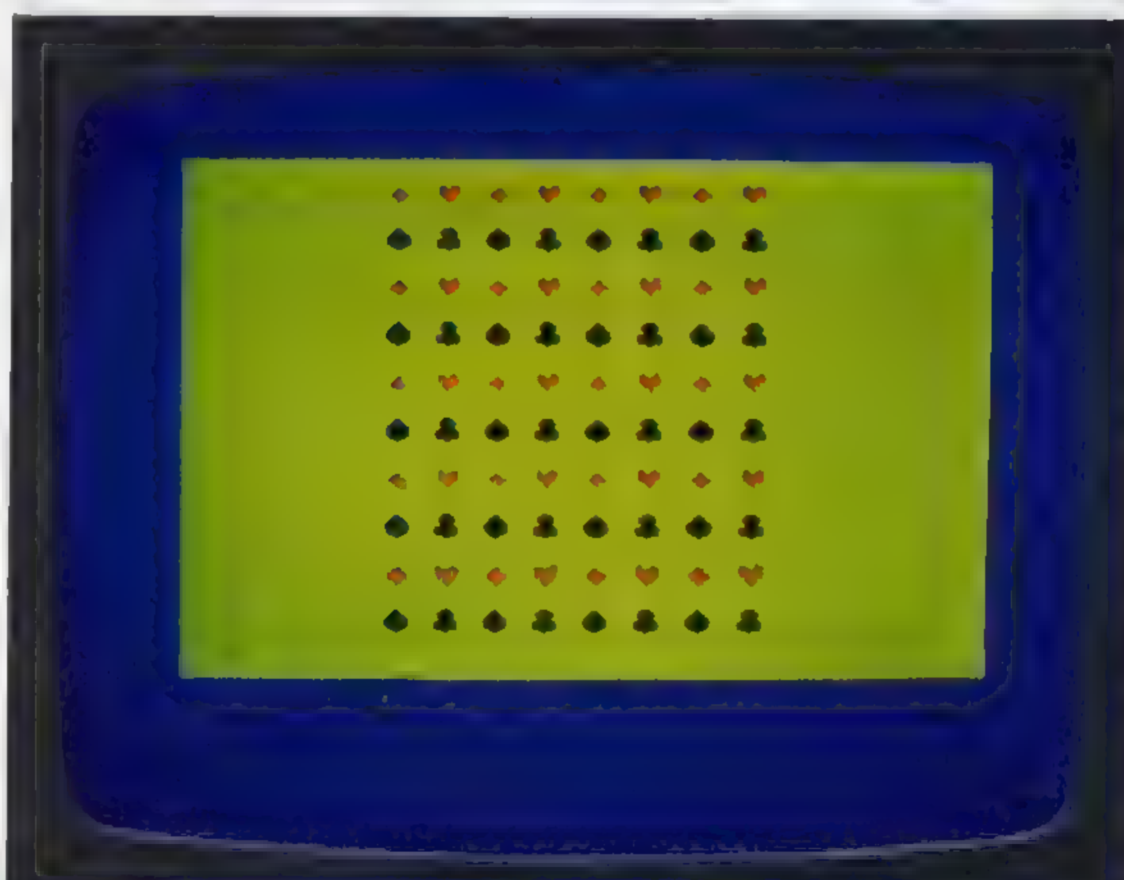
You can see from this that READ ... DATA cuts the number of lines needed from eight to five. But although this is a useful saving, the great advantage of the READ ... DATA technique is that it actually saves you far more lines if you want to program a symbol that is made up from a number of characters, or if you want to use more than one symbol in a program. Then the savings become really worthwhile.

Storing groups of characters

Here is a program that reprograms four keys – a, b, c and d – to produce the symbols for four playing-card suits. Each of the symbols is produced by one loop. With a few extra lines you can create a display with the symbols:

READ...DATA WITH 4 LOOPS





This way of producing the four symbols uses a total of 16 lines. This is eight lines less than the simple but laborious method which uses eight POKE ... USR statements per character. However, you can save even more space by READing all the DATA in the program in one FOR ... NEXT loop. Here is what the program would look like:

READ...DATA WITH 1 LOOP

At first sight, all the DATA seem to have changed. However, all that has really happened is that the DATA numbers have been keyed in a different order. The DATA is still held in the first four lines – 10 to 40. Line 60 tells the computer to READ this DATA in groups of four numbers, w,x,y,z, and each of these numbers reprograms a separate key. So the numbers that reprogram the A key are first, fifth, ninth, thirteenth, and so on. The key DATA are arranged like this:

DATA a,b,c,d,a,b,c,d,a,b,c,d ...

All you have to do to use this with your own symbols is to add up your grid totals, as in the diagram on page 30, and then make them into DATA lines by putting them together in the correct order. Now instead of saving eight lines of program, you will have saved thirteen.

How to use binary numbers with graphics

Unless you use a calculator, programming graphics characters can be quite a test of your ability at adding up. The code numbers for each column of squares in the character grid are not the most convenient numbers to deal with, and it is quite easy to make mistakes. But the Spectrum does let you key them in in another way, using the command BIN, which stands for BINary.

All computers use a binary system of electronic pulses to carry information. The word binary means that there are only two types of pulse – "on" or 1, and "off" or 0. All your programs are simply a stream of these electronic pulses. Every number that you type into the Spectrum is converted into a series of these "on" or "off" pulses, but because humans are not very familiar with the binary system, the computer is designed to accept "ordinary" numbers.

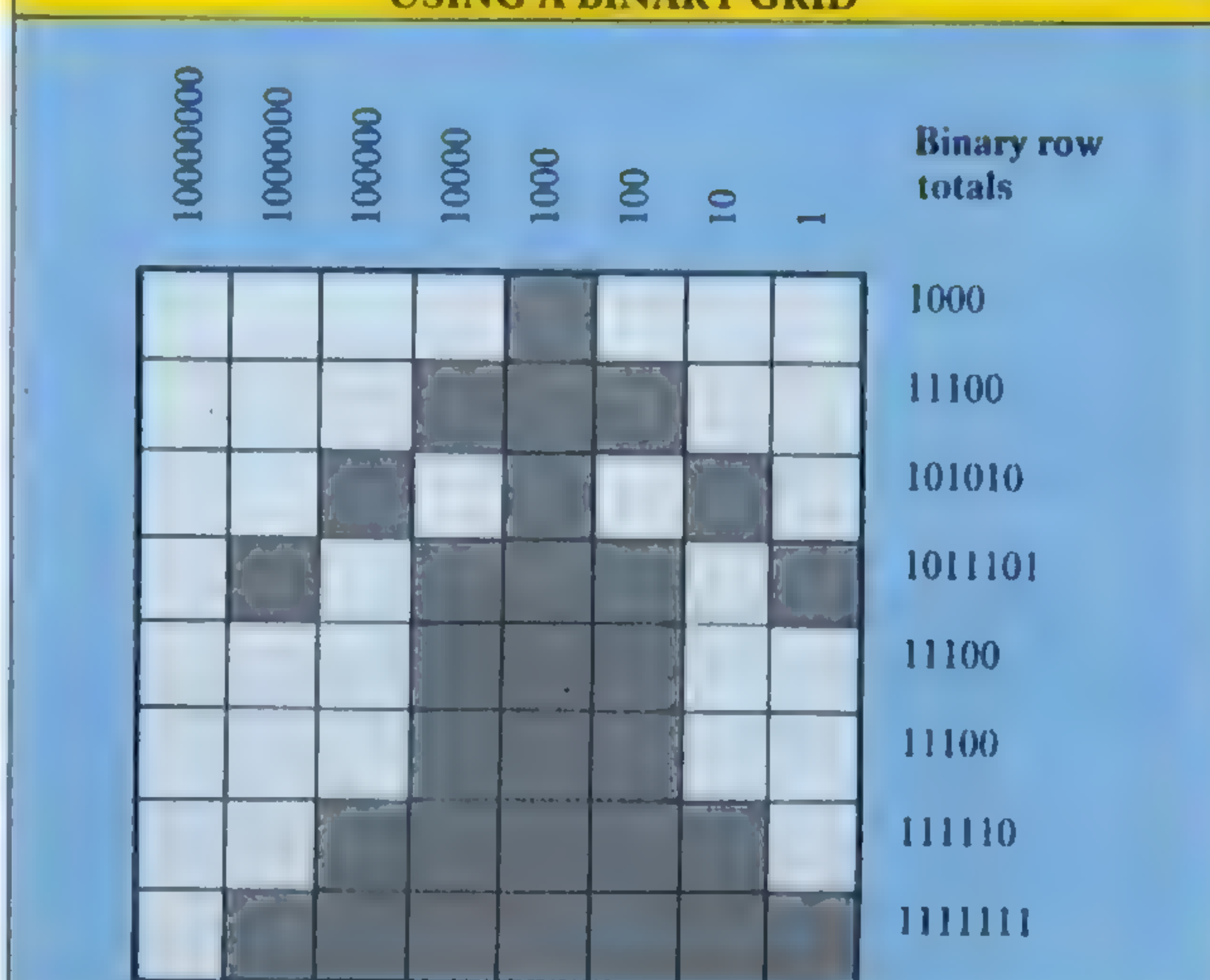
The command BIN by-passes this process and lets you enter binary numbers directly. BIN precedes a number that is written in multiples of two, instead of multiples of ten. Reset the computer and see what happens when you key in the following:

```
PRINT BIN 10
PRINT BIN 100
PRINT BIN 1011
```

Instead of seeing 10, 100 and 1011 on the screen, you should see 2, 4 and 11. The computer has converted the binary numbers back into "ordinary" numbers. The computer's ability to accept binary numbers is not of great value in simple programming. However, you can use the BIN command to save yourself having to do calculations when reprogramming keys.

You will remember that when you use a character grid, the numbers across the top of the grid go up in jumps. Each number is twice the one to its right. The computer actually gives each column a binary code, 0, 10, 100, 1000 and so on, so you can use binary numbers to enter the way a character is to be made up. Here is a grid numbered in this way:

USING A BINARY GRID



If you imagine that each filled-in square has a value of 1, and each empty square has a value of 0, you can enter the screen totals as sequences of ones and zeros using the BIN keyword or simply use BIN in a series of direct commands to PRINT each total in decimal. The great advantage of using BIN is that although the binary numbers look long, there is no adding up to be done. You simply key in what you see, counting black as 1, and white as 0.

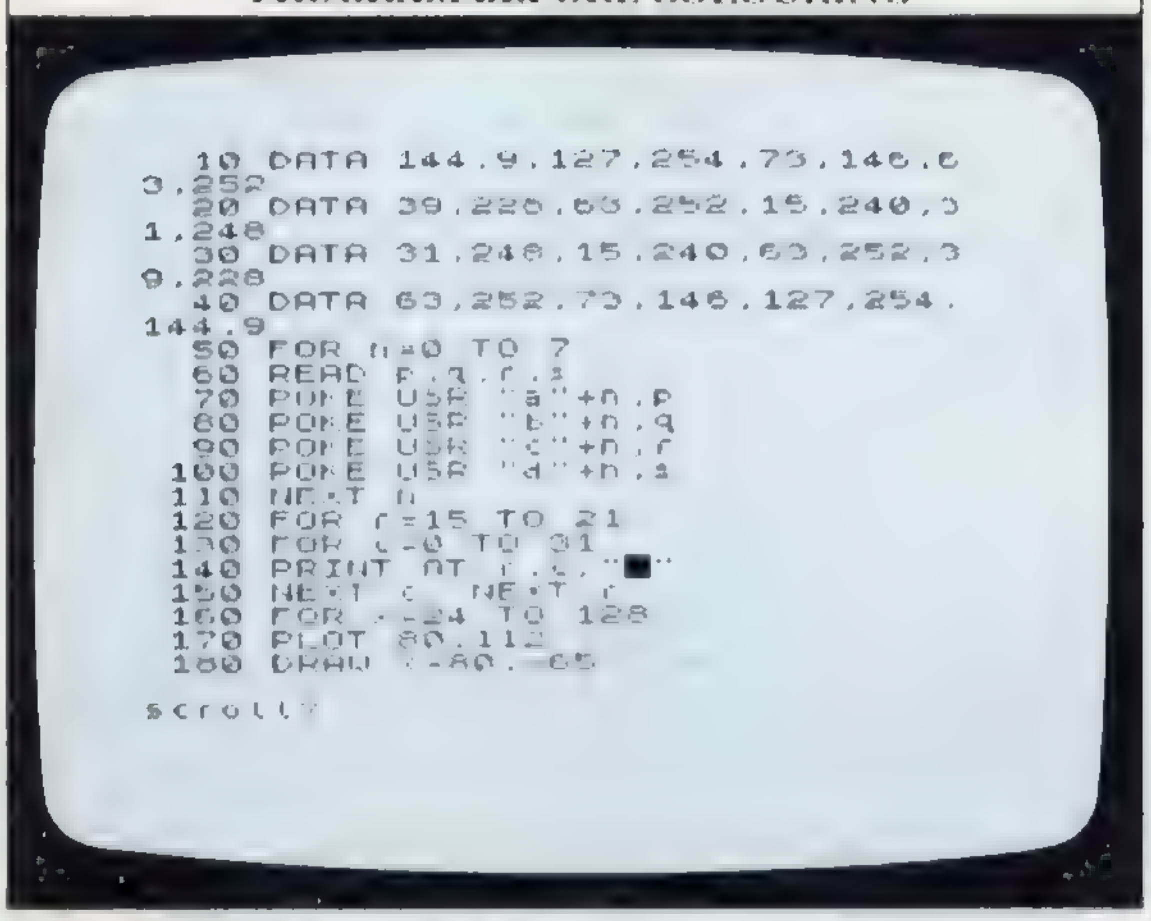
ADVANCED COLOUR GRAPHICS

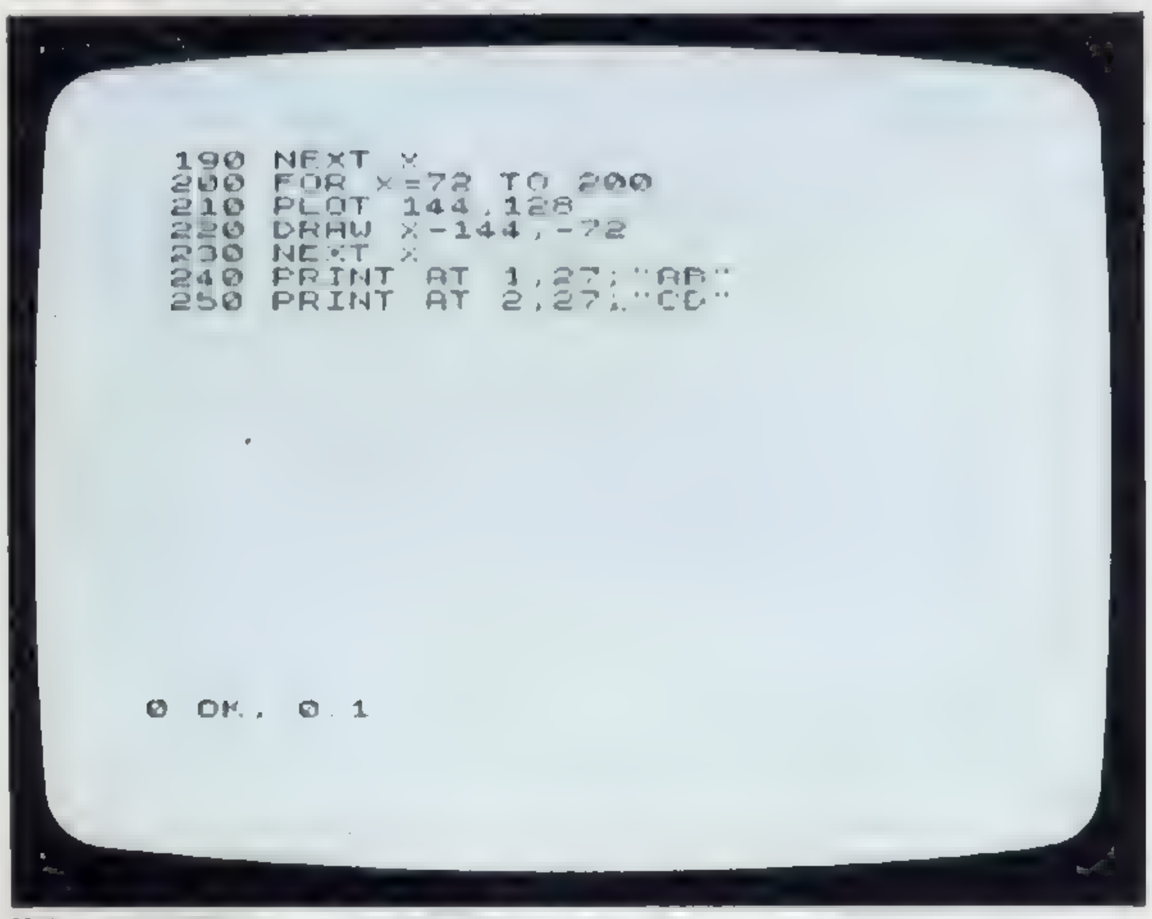
Having tried some simple colour graphics on pages 36–37, you can now move on to putting all the graphics, colour and animation commands together in one program to experiment with the ways they interact with one another. The program on these two pages produces a complex picture; if you work through this listing, you should be able to write a similar program yourself.

Creating a landscape

The first step in producing a display is to program it in black and white. Here is a listing that does that, using the "scrambled" DATA system that is dealt with on the previous two pages, and also using PLOT and DRAW to fill in two "pyramids":

PROGRAM BEFORE COLOURING





The DATA lines (10–40) define four characters that are PRINTed together by lines 240 and 250 to produce the Sun. Lines 50 to 110 transform the information in the DATA lines into user-defined characters that are controlled by keys a, b, c and d. Lines 120 to 150 PRINT seven lines of the graphics character on the 8 key to form the foreground. Lines 160 to 230 then DRAW the two pyramids as simple triangles.

Once you have keyed in all the program, RUN it to make sure that you have typed it correctly. If the following display appears, you are ready to move on to the next stage:

DISPLAY BEFORE COLOURING

Programming colour and perspective

Now it is a simple matter to add the colour:

```
  5 BORDER 0:PAPER 1:CLS
140 PRINT INK 2;AT r,c,"■"
235 INK 6
```

When you RUN the program, the changes should result in the BORDER turning black and the PAPER blue, followed by a red foreground and yellow Sun.

You can now add the perspective lines, to give the display a feeling of "depth". The lines should produce the same effect as parallel paths disappearing into the distance. You need to DRAW lines between two horizontal rows of co-ordinates, with the top points being closer together than the bottom ones. Try keying in these lines:

```
260 LET w=8: INK 7
270 FOR x=72 TO 184 STEP 16
280 PLOT x,38
290 DRAW w−x,−38
300 LET w=w+34
310 NEXT x
```

The x co-ordinate of the top (most distant) end of each line is given by line 270. STEP 16 makes x increase by 16 each time instead of 1. The graphics cursor is then

moved to the top of the line by PLOT x,38. Each line is DRAWn from that point to the bottom of the screen by line 290. Then, to make the x co-ordinate of the bottom of the line (w–x) step across the screen in bigger steps than the top, line 300 increases w by 34. Once you have done this, RUN the program again:

"CHUNKY" GRAPHICS DISPLAY

As you will see, it produces odd results. The lines are DRAWn as large squares. What has gone wrong?

DRAWing lines on colour

The answer is that although the Spectrum's graphics use 256×176 pixels, INK colour resolution is only 32×22, the same as the text resolution. These large squares are known as "chunky graphics". As the lines are DRAWn, they automatically change the colour of every square they pass through. However, if you watch the top half of the screen as the picture forms, you will see that the lines used to DRAW the pyramids do not produce these chunky graphics. One further change is needed to create this complete non-chunky display:

CORRECTED DISPLAY

The problem is that lines can be DRAWn successfully on background PAPER colour but not on INK.

You must therefore rewrite the program so that the red ground is DRAWn as a background PAPER colour – not as INK. You can do this simply by changing line 140 to PRINT with red PAPER:

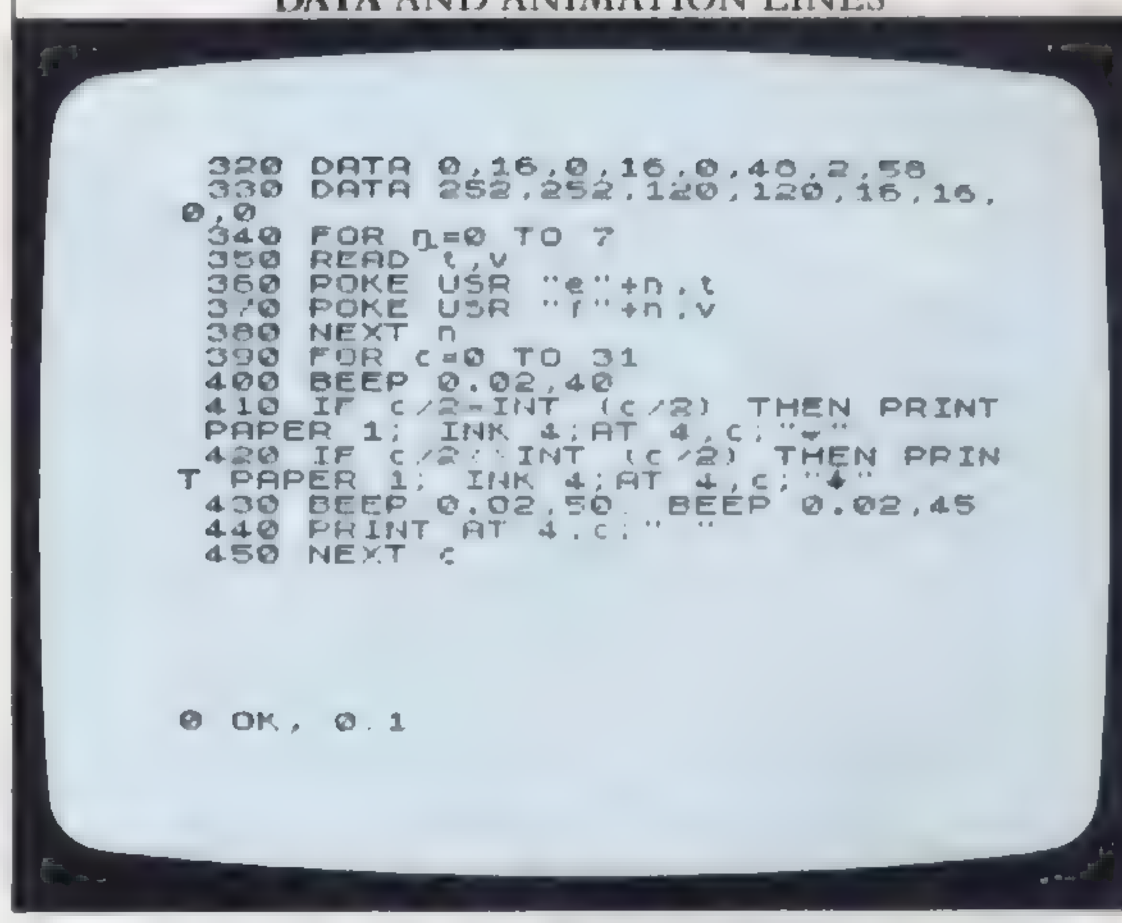
```
140 PRINT PAPER 2;AT r,c;" "
```

Now the perspective lines will appear successfully.

Adding a moving character

For the final touch, you could add a bird flying across the screen. You can produce this by using two user-defined characters, and alternating them to simulate the bird's flapping wings:

DATA AND ANIMATION LINES



```
 320 DATA 0,16,0,16,0,40,2,58
 330 DATA 252,252,120,120,16,16,
0,0
 340 FOR n=0 TO 7
 350 READ t,v
 360 POKE USR "e"+n,t
 370 POKE USR "f"+n,v
 380 NEXT n
 390 FOR c=0 TO 31
 400 BEEP 0.02,40
 410 IF c/2=INT (c/2) THEN PRINT
 PAPER 1; INK 4;AT 4,c;"▾"
 420 IF c/2<>INT (c/2) THEN PRIN
T PAPER 1; INK 4;AT 4,c;"♦"
 430 BEEP 0.02,50: BEEP 0.02,45
 440 PRINT AT 4,c;" "
 450 NEXT c



0 OK, 0:1
```

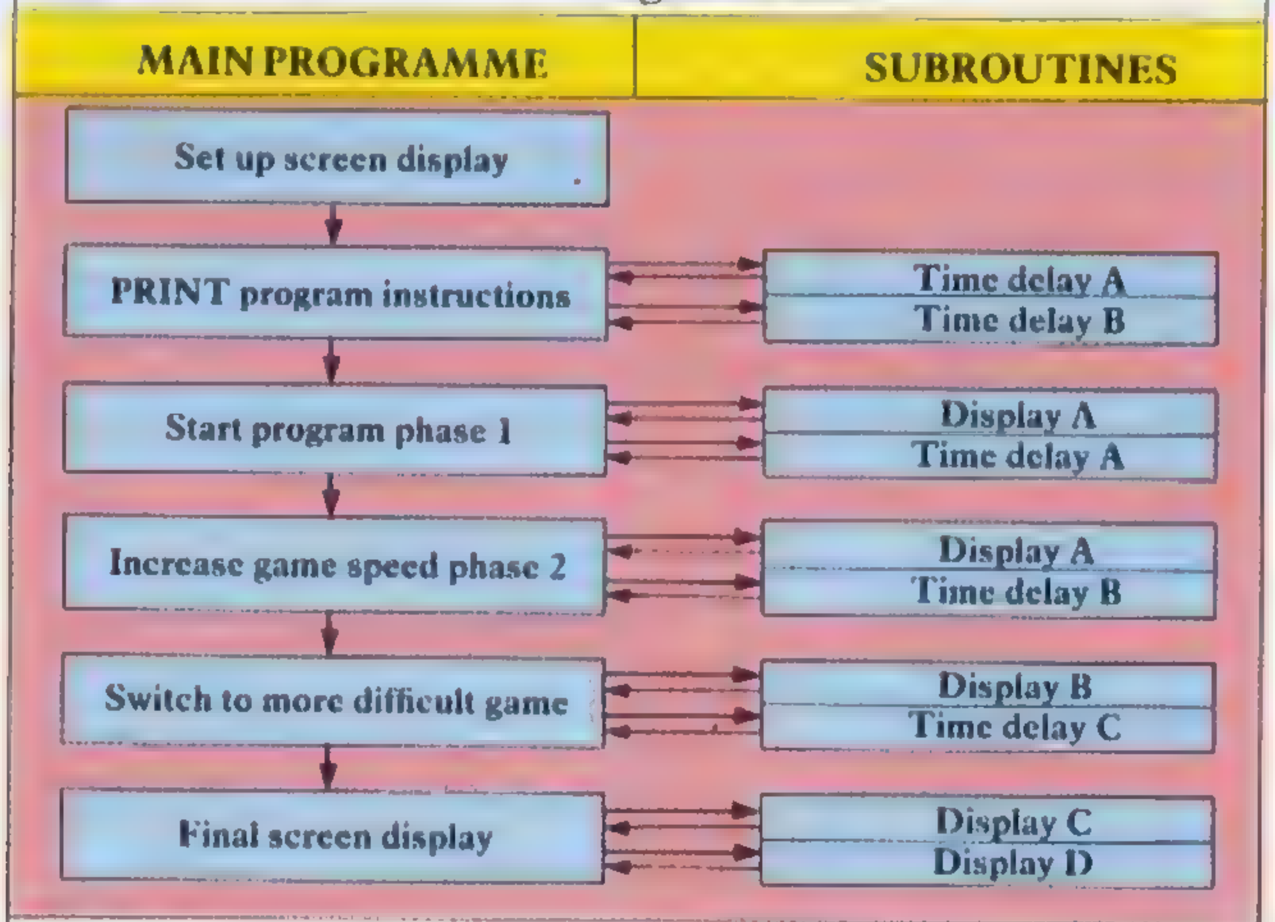
Lines 410 and 420 use IF and THEN to make the computer check whether c, the bird's column position, is even or odd. If it is even, the bird's body alone is PRINTed, if it is odd, then a flapping wing is added. The display is now complete:

FINAL DISPLAY

WRITING SUBROUTINES

You will often want to use the same few lines of a program again and again to carry out the same calculation or to display the same group of characters on the screen. To avoid writing out the same lines time after time (and using up too much of the computer's memory) you could branch off to frequently-used sections of the program with GOTO. However, relying on GOTO is frowned on by many programmers. Using it carelessly can turn your programs into untidy mazes that are impossible to understand or debug.

The easiest program to analyze and debug is one that is written methodically in blocks or modules, each of which you can test independently of the others, if problems arise. If you look up the listing of a good games program in a magazine, for example, you will find that it works something like this:



How to use a subroutine

Frequently-used blocks of programs, or subroutines, are written using the command GOSUB. This allows you to branch off from the main program to the subroutine and then return to the main program again. The command looks like this:

```
50 GOSUB 500
```

Here the main program RUNs normally until it reaches line 50, which makes the computer jump to a subroutine at line 500. After it has been through the subroutine, it returns to the main program at line 60 – the one after the line where it left. The subroutine must be ended by the word RETURN. Without it, the computer will not go back to the correct point in the main program.

You can use GOSUB in almost any program where the computer has to repeat an operation. The next program produces a temperature conversion chart, using three types of measurement, Centigrade, Fahrenheit and Kelvin. The subroutine at line 80 makes the computer PRINT out a line on the table, produce a short BEEP, and then return to line 60. The command STOP at line 70 stops the program carrying on into the subroutine. If you miss out STOP, the computer will reach the RETURN command at line 110. It would then produce an error report because it had encountered a RETURN without its own GOSUB. You will notice that the subroutine in the listing below is inside a FOR ... NEXT loop, so it is "called" a number of times. The display produced when you RUN this program is shown on the bottom screen:

TEMPERATURE CONVERSION PROGRAM

```
  10 BORDER 1  PAPER 2  INK 7. C
LS
  20 PRINT AT 1,4;"C";AT 1,15;"F
";AT 1,26;"K"
  30 PRINT "--------------------
--------------"
  40 FOR c=-30 TO 140 STEP 10
  50 GO SUB 80
  60 NEXT c
  70 STOP
  80 PRINT TAB 3;c;TAB 14;(c*9/5
)+32;TAB 25;c+273
  90 BEEP 0.05,40
 100 PAUSE 25
 110 RETURN

0 OK, 0:1
```

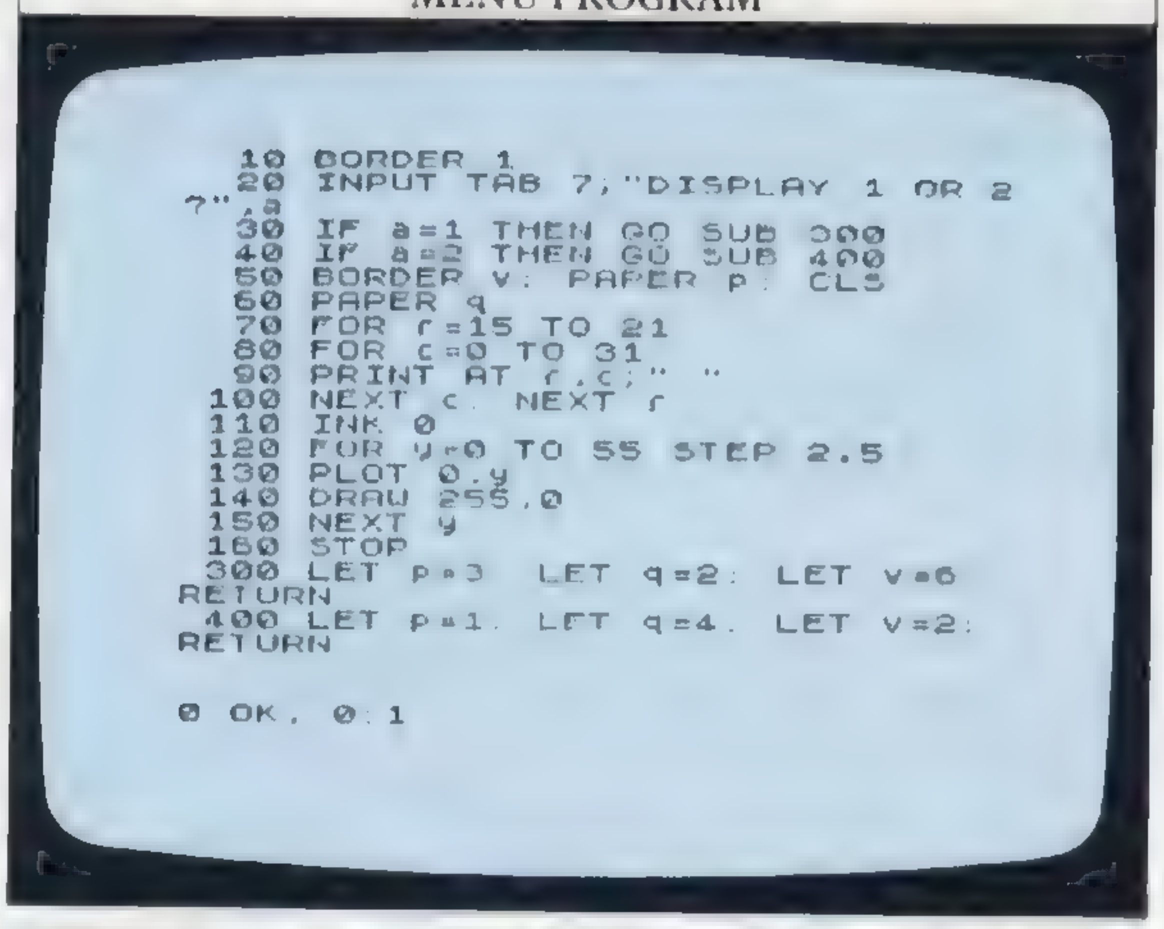
In this program, the subroutine is not actually saving any space. However, if you extended the program to carry out other functions, the subroutine could be "called" again as often as you wanted – saving both space and memory.

Setting up displays with GOSUB

In many programs, you are initially given a "menu" or choice of options to select. This choice is often programmed by using GOSUB. When you enter your selection, the program goes to the appropriate subroutine and sets up the display you have picked.

Here is a simple listing that shows how you can do this. The program can set up either of two basic displays. One is illustrated on the bottom screen. The colours in each display are produced by a subroutine – which subroutine is used depends on your INPUT following line 20. If you were using this subroutine in a real games program, you could use these colour settings often by putting in a GOSUB:

MENU PROGRAM



```
  10 BORDER 1
  20 INPUT TAB 7;"DISPLAY 1 OR 2
?",a
  30 IF a=1 THEN GO SUB 300
  40 IF a=2 THEN GO SUB 400
  50 BORDER v: PAPER p: CLS
  60 PAPER q
  70 FOR r=15 TO 21
  80 FOR c=0 TO 31
  90 PRINT AT r,c;" "
 100 NEXT c: NEXT r
 110 INK 0
 120 FOR y=0 TO 55 STEP 2.5
 130 PLOT 0,y
 140 DRAW 255,0
 150 NEXT y
 160 STOP
 300 LET p=3: LET q=2: LET v=6:
RETURN
 400 LET p=1: LET q=4: LET v=2:
RETURN

0 OK, 0:1
```

Using GOSUB with animation

Here is a program which PRINTs a target – two graphics symbols on the ground – and which then PRINTs a succession of aliens falling from random points at the top of the screen. If one of the aliens then hits the target, line 240 directs the computer to go to the subroutine at line 270. The subroutine produces a sequence of changing BORDER colours while BEEPing, and then replaces the target, restarting the program. The bottom screen shows the display:

GOSUB ANIMATION PROGRAM

```
  10 DATA 0,0,64,32,18,11,13,31
  20 DATA 0,0,2,4,72,208,176,248
  30 DATA 59,25,15,63,79,95,129,
131
  40 DATA 220,152,240,252,242,25
0,129,193
  50 FOR n=0 TO 7: READ p
  60 POKE USR "a"+n,p
  70 NEXT n
  80 FOR n=0 TO 7: READ q
  90 POKE USR "b"+n,q
 100 NEXT n
 110 FOR n=0 TO 7: READ r
 120 POKE USR "c"+n,r
 130 NEXT n
 140 FOR n=0 TO 7: READ s
 150 POKE USR "d"+n,s
 160 NEXT n
 170 BORDER 1: PAPER 1: CLS
 180 PRINT INK 2;AT 21,8;"▀▀"
 190 LET x=2+INT (RND*15)
 200 FOR f=0 TO 19

scroll?
```

```
 210>PRINT AT f,x;"  "
 220 PRINT INK 6;AT f+1,x;"AB"
 230 PRINT INK 6;AT f+2,x;"CD"
 240 IF f=18 AND x=8 THEN GO SUB
270
 250 NEXT f
 260 GO TO 180
 270 FOR t=1 TO 30
 280 BEEP .005,30+t
 290 FOR g=0 TO 7
 300 BORDER g: NEXT g
 310 NEXT t
 320 BORDER 1: LET f=19
 330 CLS : RETURN

0 OK, 0:1
```

HINTS AND TIPS

When you are learning to program your Spectrum, you will have come across a number of ways of improving your technique by trial and error. However, there are some ways of saving time or sorting out problems which, although simple and effective, are not necessarily obvious. On these two pages you will find some "tricks" which will help you to produce programs that are well organized and bug-free.

Using REM as a marker or mask

Because the REM command makes the computer ignore anything that follows it in a line, it can be used in labelling and testing parts of a program. On page 18 you saw how REM can be used in the first line of a program to show you what a program does, and the longer a program is, the more useful this becomes.

However, when a program gets really long, it is sometimes difficult to pick out the REM lines among all the others. One way you can draw attention to them is by following REM with some symbols which clearly stand out from the rest of the program. Here is one way of doing this:

MARKER LINES WITH **REM**

```
  10 REM ************************
  20 REM COLOUR PYRAMID PROGRAM
  30 REM © Ian Graham 1984
  40 REM ************************
  50 BORDER 0  PAPER 1: CLS
  60 DATA 144,9,127,254,73,146,6
3,252
  70 DATA 39,228,63,252,15,240,3
1,248
  80 DATA 31,248,15,240,63,252,3
9,228
  90 DATA 63,252,73,146,127,254,
144,9
 100 FOR n=0 TO 7
 110 READ p,q,r,s
 120 POKE USR "a"+n,p
 130 POKE USR "b"+n,q
 140 POKE USR "c"+n,r
 150 POKE USR "d"+n,s
 160 NEXT n
 170 FOR r=15 TO 21
 180 FOR c=0 TO 31

scroll?
```

When you read through this program, the REM lines are visible at a glance.

REM also has a use in program development. You will often want to test a program to see what happens if certain lines are left out. This may be because part of a program takes a long time to RUN, or produces a BEEP that you don't want to hear time and time again.

You can skip part of a program by using GOTO or RUN followed by a line number, but this won't help if you just want to miss out a few lines in the middle. The way to deal with this problem without deleting the lines is to insert a REM command at the beginning of each line you want to skip. This will mask or "disable" the lines, as the computer will ignore all the commands following each REM. Here is a program in which this has been done:

LINE DISABLED WITH **REM**

```
  10 BORDER 1. PAPER 2. INK 7. C
LS
  20 PRINT AT 1,4;"C";AT 1,15;"F
";AT 1,26;"K"
  30 PRINT "---------------------
-----------"
  40 FOR c=-30 TO 140 STEP 10
  50 GO SUB 80
  60 NEXT c
  70 STOP
  80 PRINT TAB 3;c;TAB 14;(c*9/5
)+32;TAB 25;c+273
  90 REM BEEP 0.05,40
 100 PAUSE 25
 110 RETURN




0 OK, 0.1
```

How to check nested loops

When you use a number of loops in a program, it is easy to get the loops tangled so that the program does not produce the results you want. There is an easy way to check whether the loops are correctly "nested" – or that they fit inside each other without overlapping.

If you note the program down (or better still, if you can print it on a printer) you can connect the start of every loop up with its end:

LINKING LOOPS

```
    10 BORDER 0: PAPER 0. CLS
 ┌──20 FOR a=0 TO 20
 │┌─30 FOR c=0 TO 7
 ││ 40 PRINT INK c,"■";
 ││ 50 PAUSE 10
 │└─60 NEXT c
 │┌─70 FOR c=0 TO 7
 ││ 80 PRINT INK c;"▞";
 ││ 90 PAUSE 10
 │└100 NEXT c
 └─110 NEXT a




0 OK, 0.1
```

Alternatively you can jot down every FOR and NEXT in a program on a piece of paper in the order in which they appear, missing out the intervening lines. It's then an easy process to link the loops up.

If all the loops are correctly nested, none of these lines should overlap. If they do, you have wrongly nested loops, and the chances are that the program will not work correctly.

Improving keyboard feedback

Every time you press a key on the Spectrum, the computer makes a click. It's very useful. Without it, you have to keep looking at the screen to make sure that every key press has been registered by the computer. However, although it is loud enough for work in quiet surroundings, it can be drowned by noise from any other source, making programming more difficult. If the keyboard click isn't loud enough, you can increase it from a click to a more distinct BEEP by typing:

```
POKE 23609,n
```

where n is a whole number from 0 to 255. Zero produces a click, 255 a long, high-pitched note.

How to speed up editing

If you want to edit a line in the middle of a long program, you can simply type in a LIST instruction like this:

```
LIST 250
```

This will bring the line indicator down to the line to be edited. However, if the LISTing continues onto a further frame, the "scroll?" prompt appears at the bottom of the screen. If you press CAPS SHIFT and EDIT, the program will just scroll onwards to the next "page". You can press N to stop the listing scrolling and then carry on with editing, but there is an alternative that eliminates "scroll?" altogether.

If you want to edit line 250 in a long program, type:

```
249
```

followed by ENTER. Nothing much seems to happen. The line marker doesn't move. But more importantly, "scroll?" doesn't appear either. If you now press CAPS SHIFT and EDIT, line 250 then appears at the foot of the screen ready to be edited. The 249 – or whichever number you choose – should be a number which immediately precedes the number of the line that you want to edit. Make sure though that the number you use is not a line number already in the program. If it is, then typing the number followed by ENTER will erase the program line from the computer's memory.

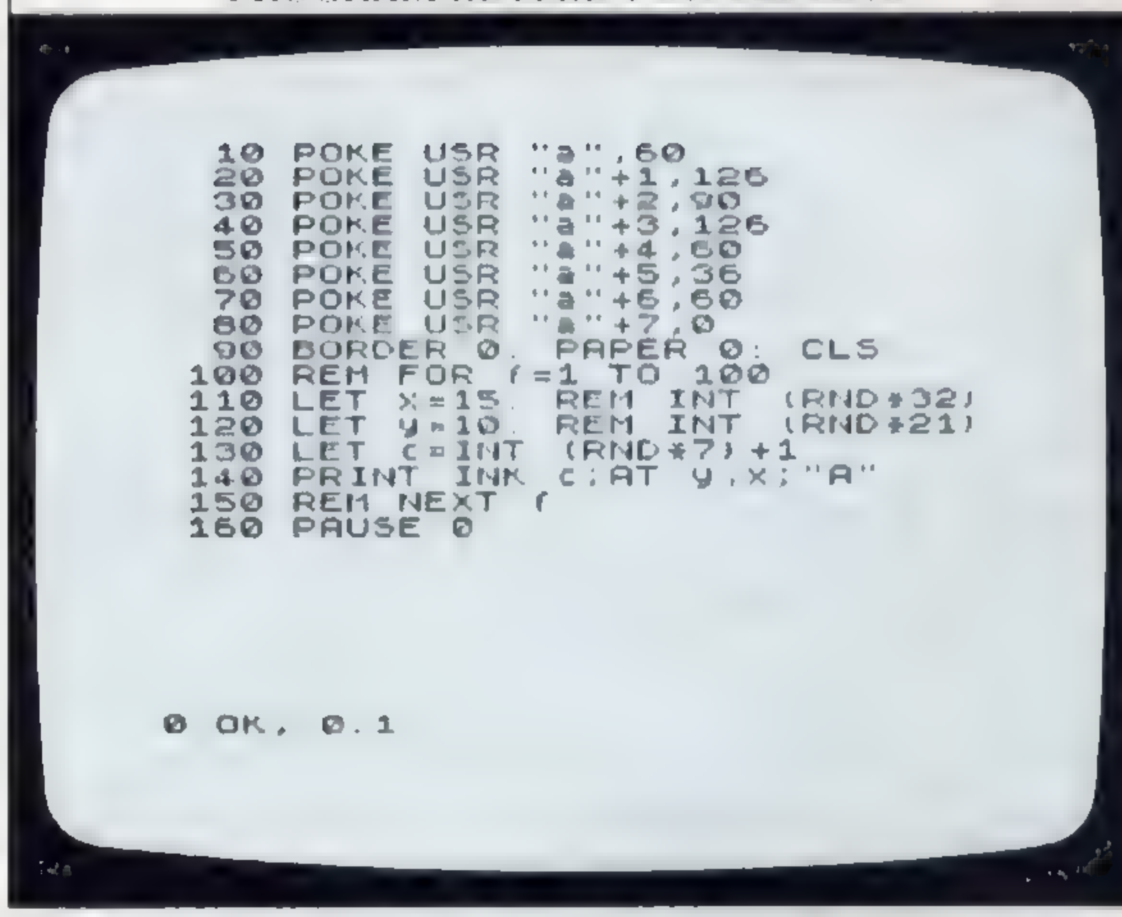
Useful debugging techniques

Although the Spectrum has a large repertoire of error reports which will alert you to any incorrect lines in a program, often a program will RUN without any hitches, only to produce a result entirely different to the one you had in mind. How then do you go about finding the source of the problem?

As you have just seen, you can use REMs to mask parts of a program, or you can link loops to check that they are nested properly. But if that doesn't help, you can often track down the problem by giving each variable in a program one set value, instead of allowing it to go through many.

Imagine that you have a graphics program which uses the command RND to produce a display which is built up by looping. If it does not work in the way you expect, you can take out the RND, and instead use a number. You can then work out what effect this number should have when the program is RUN. Now take out the lines that start and terminate the loop (you can use REM for this). If the result of a single RUN through is not what you predicted, the display should give you some idea of where your program is going "wrong":

PROGRAM EDITED FOR TESTING



Above is the random graphics program from page 49, edited so that the random variables in lines 110 and 120 are fixed. The original lines are still kept in, but are disabled by REMs. The loop between lines 100 and 150 is also disabled by a pair of REMs so the program only PRINTs once.

If the program is RUN, you can check whether or not the program has done what was expected, and if not, it is now much easier to work backwards to the source of the problem. You can use this technique in any program which uses variables. By substituting a single value for each variable, you can check your expected result with the result when the program is RUN.

Finally, don't forget that the BREAK key can be very helpful in telling you how far the computer has got through a program. If you RUN a program which either seems to do nothing, or gets stuck at a certain point, the BREAK key will tell you where the hold-up lies. If you then LIST the program, you will often be able to identify the problem with the line identified by BREAK and correct it.

HOW TO KEEP YOUR PROGRAMS

Whatever you type into your Spectrum is only stored in the computer's memory as long as it is supplied with power. When you switch the computer off, your program disappears. Obviously, you can't type in every program you want to use afresh each time you switch on the computer. Fortunately, you probably already have a means of storing programs cheaply and easily – an ordinary tape cassette recorder. The Spectrum has two sockets on its rear panel, labelled EAR and MIC, and these should be connected to the corresponding recorder sockets.

Setting the cassette controls

Having connected a recorder to the computer, the next job is to set the volume and tone controls properly. If there is a tone control, set it to maximum treble. The volume control may need a little more experimentation. Set the volume control midway between minimum and maximum.

Programs are recorded on tape and loaded back into the computer using two commands – SAVE and LOAD. You can test these commands by trying to SAVE any program from this book. Type the program into the computer again. RUN it to make sure that there are no typing errors and then unplug the EAR connection. Now give it any filename you like and ask the computer to SAVE it:

SAVE "FOR...NEXT"

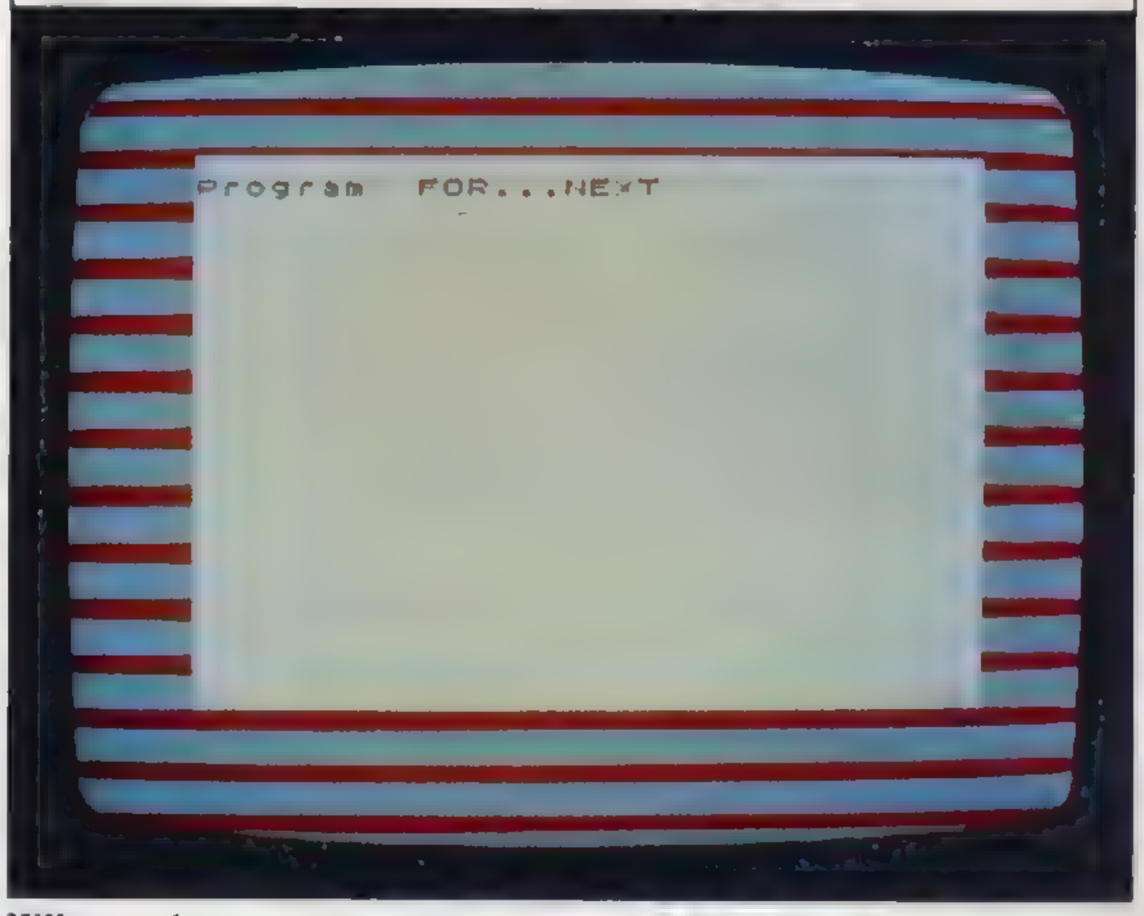
The computer will reply with:

Start tape then press any key

Press RECORD and PLAY on the tape recorder and then press one of the computer keys. Two types of screen pattern will appear:

CASSETTE SCREEN DISPLAY

CASSETTE SCREEN DISPLAY



When the program has been SAVEd, the pattern of lines disappears and the OK prompt reappears at the bottom of the screen. Stop the tape.

Checking a recording

To check that the program has been recorded properly, reconnect the EAR lead and type:

VERIFY "FOR...NEXT"

Start the tape playing again. As each program recorded on the tape is played back into the computer, the pattern of red and blue stripes should reappear on the screen, followed by the program's name. If not, then the program has not been SAVEd properly. Rewind the tape, turn up the volume and press PLAY. If you still cannot VERIFY the program, interrupt VERIFY by pressing CAPS SHIFT plus BREAK. Try recording the program again at a higher volume setting (don't forget to remove the EAR plug).

You can use VERIFY to catalogue all the programs on a tape. Type VERIFY "cats" (or any other filename that you know you have not used). As each program on the tape is read by the computer, its filename will appear on the screen.

Playing back a program

Now try LOADing the program back into memory. Type in the filename like this:

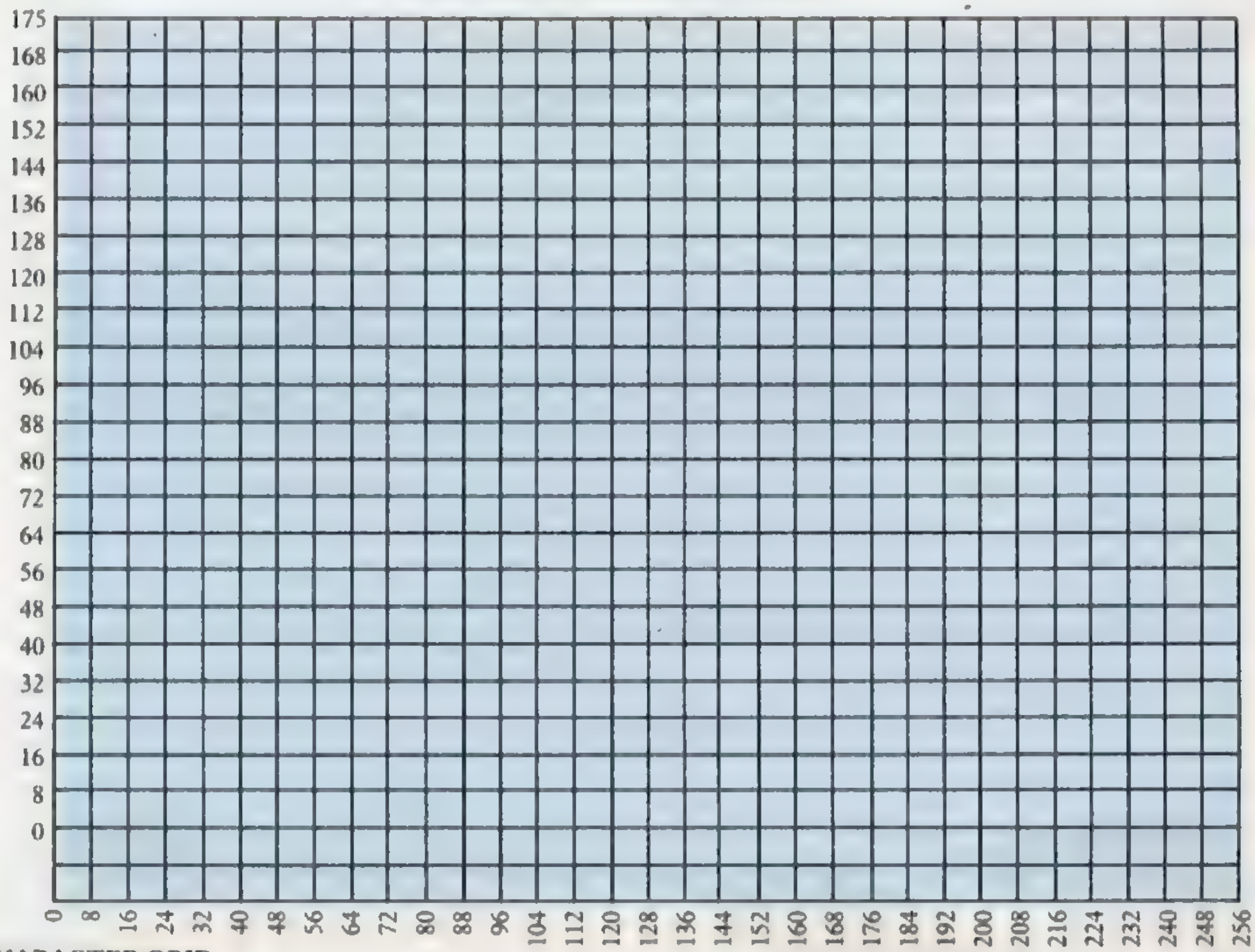
LOAD "FOR...NEXT"

Make sure the tape is rewound. Start it playing. When the computer finds the program, "program:" followed by the filename will appear on the screen. When the program is fully LOADed, the OK screen prompt reappears. Now you can RUN the program.

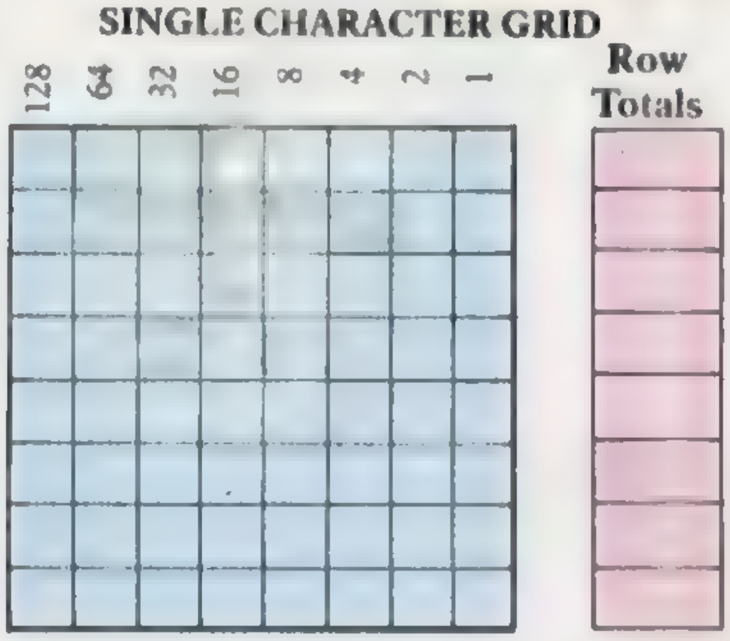
GRAPHICS AND CHARACTER GRIDS

The grid below shows the co-ordinates of the screen display when graphics commands are used. A point on the screen is identified by two co-ordinates x,y. The first co-ordinate sets the horizontal position which is measured along from the left-hand side of the screen. The second co-ordinate sets the vertical position measured from the bottom of the screen. A character PRINTed on the screen occupies an area that is 8 graphics units wide and 8 graphic units high. You cannot PRINT on the bottom two lines of the screen.

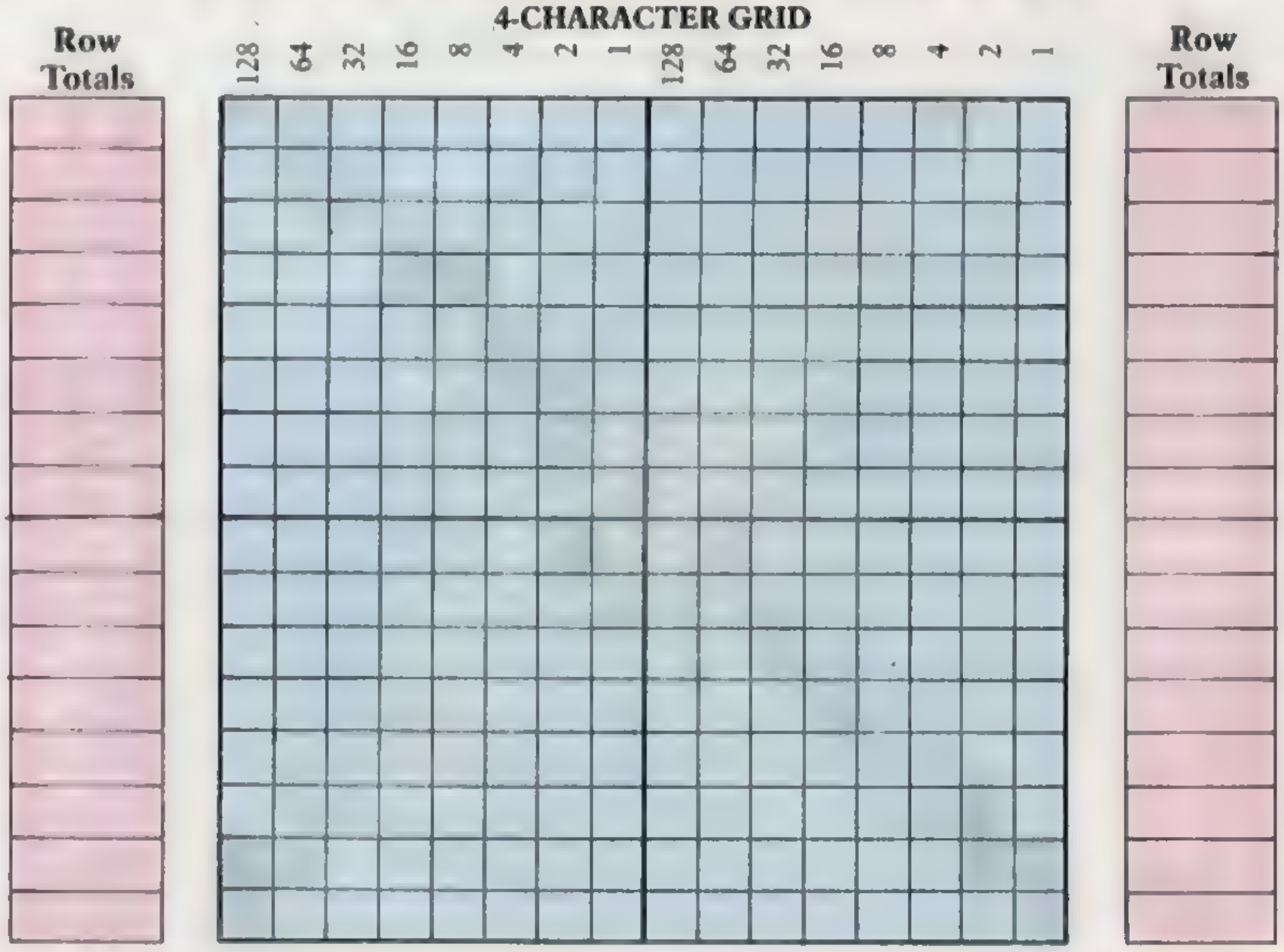
GRAPHICS CO-ORDINATES GRID



SINGLE CHARACTER GRID



4-CHARACTER GRID



Character grids These grids can be used to design either a single character *(above)* or a symbol made from up to four characters *(right)*. You can pencil in your design on the grids and then use the blue columns to list the row totals. These are used in a program with the commands POKE USR. Keys A to U are free for programming user-defined characters.

GLOSSARY

Entries in **bold type** are BASIC keywords.

AT
Used with **PRINT** to place characters on the screen.

BASIC
Beginners' All-purpose Symbolic Instruction Code; the most commonly used high-level programming language.

BEEP
Makes the computer sound a short note or beep, whose duration and pitch are determined by the numbers following the command.

BIN
Converts a number written in binary into the equivalent number written in decimal.

Bit
A binary digit – either 0 or 1.

BORDER
Changes the colour of the screen's border area.

BRIGHT
Turns specified characters to a brighter shade of their **INK** colour.

Byte
A group of eight bits.

Chip
A single package containing a complete electronic circuit. Also called an integrated circuit (IC).

CLS
Clears the text area of the screen.

CPU
Central Processing Unit. Normally contained in a single chip called a microprocessor, this carries out the computer's arithmetic and controls operations in the rest of the computer.

Cursor
A flashing symbol on the screen, showing where the next character will appear.

DATA
Stores a line of data. The computer treats whatever follows **DATA** as information that may be needed later in the program. Used in conjunction with **READ**.

DRAW
Draws a line in the current **INK** colour from the graphics origin at 0,0 or the last point visited to a point specified.

FLASH
Makes characters flash on the screen.

Flowchart
A diagrammatic representation of the steps necessary to solve a problem.

FOR...NEXT
A loop which repeats a sequence of program statements a specified number of times.

GOSUB
Makes the program jump to a subroutine beginning at the line number following the command. The subroutine must always be terminated by **RETURN**.

GOTO
Makes a program jump to the line number following the command.

Hardware
The physical machinery of a computer system, as distinct from the programs (software).

IF...THEN
Prompts the computer to take a particular course of action only if the condition specified is detected.

INK
Changes the colour of text and graphics that appear on the screen.

INPUT
Instructs the computer to wait for some data from the keyboard which is then used in a program.

INT
Converts a number with a decimal fraction into a whole number.

Interface
The hardware and software connection between a computer and another piece of equipment.

INVERSE
Switches the **PAPER** colour for the **INK** colour and vice versa.

K
Abbreviation of kilobyte (1024 bytes).

LET
Assigns a value to a variable.

LIST
Makes the computer display the program currently in its memory.

LOAD
Transfers a program from a cassette tape into the computer's memory.

Loop
A sequence of program statements which is executed repeatedly or until a specified condition is met.

NEW
Removes a program from the computer's memory so that a new program can be keyed in.

OVER
Allows new characters to be **PRINT**ed on top of existing characters without erasing the existing characters.

PAPER
Changes the screen's background colour.

PAUSE
Halts a program for a period set by a number measured in fiftieths of a second.

PLOT
Makes a single dot appear on the screen at the point specified by the co-ordinates that follow it.

POKE USR
Stores a number that reprograms a key to produce a user-defined character.

PRINT
Makes whatever follows appear on the screen.

RAM
Random Access Memory (volatile memory). A memory whose contents are erased when the power is switched off. See also ROM.

READ
Instructs the computer to take a specific number of items from a **DATA** statement so that they can be used in a program.

REM
Enables the programmer to add remarks to a program. The computer ignores whatever follows **REM** in a program statement.

RESTORE
Resets the point from which **DATA** items are **READ**, so that items can be used more than once in a program.

RETURN
Terminates a subroutine. (See also **GOSUB**).

RND
Produces numbers between 0 and 1 at random which can be used to produce unpredictable sequences.

ROM
Read Only Memory (non-volatile memory). A memory which is programmed permanently by the manufacturer and whose contents can only be read by the user's computer.

SAVE
Records a program currently in the computer's memory onto a tape cassette. The program is identified by a filename.

Software
Computer programs.

SQR
Produces the square root of the number that follows it.

STEP
Sets the step size in a **FOR...NEXT** loop.

STOP
Halts a program and **PRINT**s out the line number in which it appears.

String
A sequence of characters treated as a single item – someone's name, for instance.

Subroutine
A part of a program that can be called when necessary, to produce a particular display or carry out a number of calculations repeatedly for example.

TAB
Used with **PRINT** to specify how far along a line characters are to appear.

Variable
A labelled slot in the computer's memory in which information can be stored and retrieved later in a program.

VERIFY
Checks that a program that is currently in memory has been recorded correctly on a tape cassette using **SAVE**.

INDEX

Main entries are in **bold type**

DK

Screen Shot

PROGRAMMING SERIES

The original and exciting new teach-yourself programming course for ZX Spectrum owners.

Over 150 unique screen-shot photographs of program listings *and* programs in action – showing on the page exactly what appears on the screen.

Packed full of programming tips and techniques, reference charts and tables, and advice on how to get the most out of your ZX Spectrum.

CONTENTS INCLUDE

Setting up and starting off • Inside your computer • Screen layout and how to control it • Computer conversations • The electronic drawing-board • DIY graphics • Animation • Special effects • Compiling a data bank

Further volumes in the *Screen Shot* **series include**

Step-by-Step Programming for the **ZX Spectrum** BOOK TWO

PLUS

Step-by-Step Programming for the **BBC Micro, Commodore 64, Acorn Electron,** and **Apple II**

DORLING KINDERSLEY

ISBN 0-86318-026-4

00595

£5.95

9 780863 180262

ScreenShot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM

IAN GRAHAM

BOOK TWO

Further techniques for exciting games and graphics – in full colour

DK

ScreenShot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

ZX SPECTRUM

THE DK SCREEN-SHOT PROGRAMMING SERIES

Never has there been a more urgent need for a series of well-produced, straightforward, practical guides to learning to use a computer. It is in response to this demand that The DK Screen-Shot Programming Series has been created. It is a completely new concept in the field of teach-yourself computing. And it is the first comprehensive library of highly illustrated, machine-specific, step-by-step programming manuals.

BOOKS ABOUT THE ZX SPECTRUM

This is Book Two in a series of unique step-by-step guides to programming the ZX Spectrum. Together with its companion volumes, it will build up into a self-contained teaching course that begins with the basic principles of programming, and progresses – via more sophisticated techniques and routines – to an advanced level.

BOOKS ABOUT OTHER COMPUTERS

Additional titles in the series will cover each of the world's most popular computers. These will include:

Step-by-Step Programming for the **BBC Micro**

Step-by-Step Programming for the **Commodore 64**

Step-by-Step Programming for the **Acorn Electron**

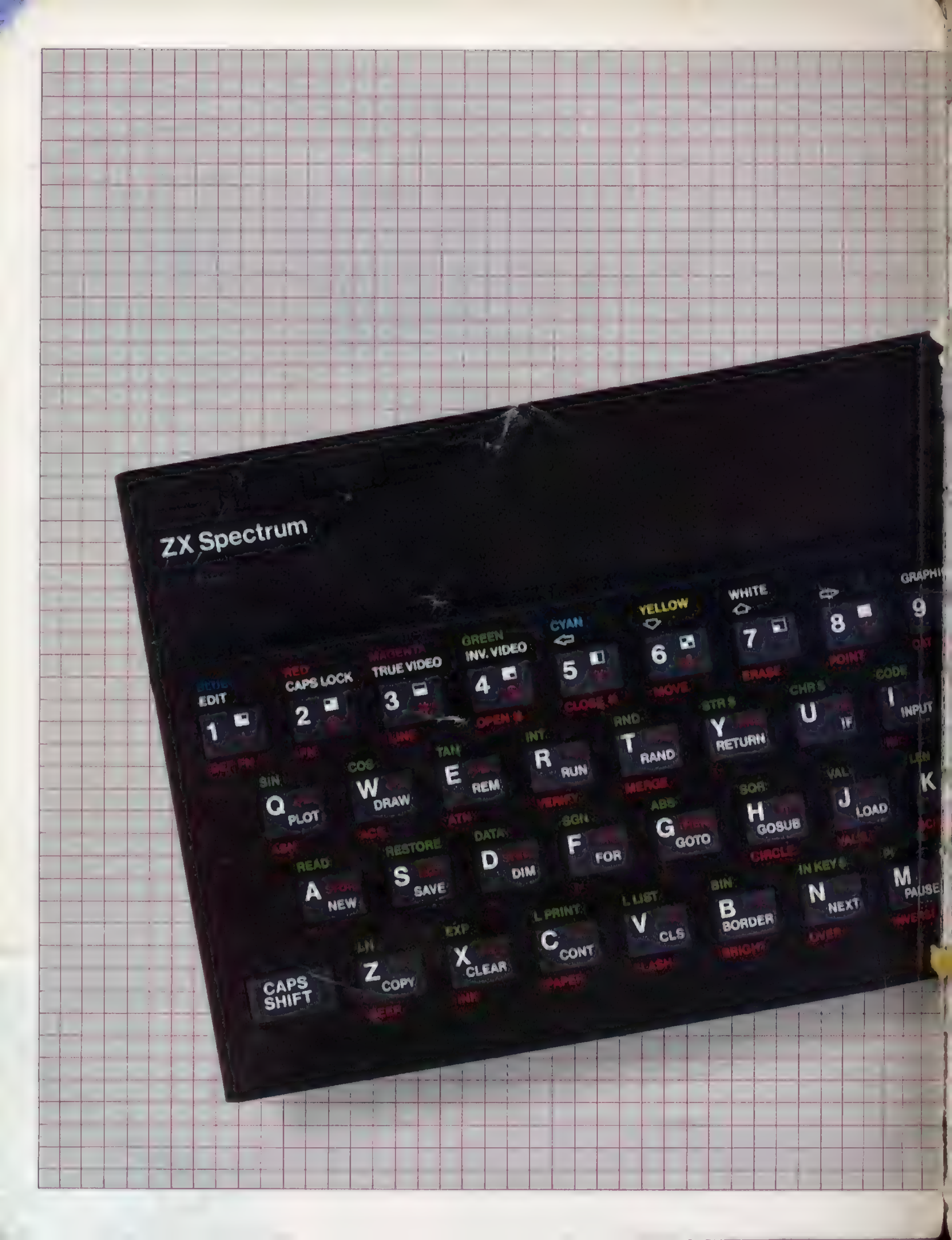
Step-by-Step Programming for the **Apple II**

Step-by-Step Programming for the **IBM PCjr**

IAN GRAHAM

After taking a B.Sc. in Applied Physics and a postgraduate diploma in Journalism at The City University, London, Ian Graham worked as assistant editor of *Electronics Today International* and deputy editor of *Which Video?* Since becoming a full-time freelance writer in 1982, he has contributed to a wide range of technical magazines (including *Computing Today, Video Today, Video Search, Hobby Electronics, Electronic Insight, Popular Hi-Fi, Science Now,* and *Next...*) and has also written a number of popular books on computers and computing. These include *Computer & Video Games, Information Technology, The Inside Story – Computers,* and *The Personal Computer Handbook* (co-written with Helen Varley).

BOOK TWO



ZX Spectrum
EDIT
CAPS LOCK
TRUE VIDEO
GREEN
INV. VIDEO
CYAN
YELLOW
WHITE
GRAPHI
1
2
3
4
5
6
7
8
9
CODE
INPUT
CHR $
U
IF
STR $
Y
RETURN
RND
T
RAND
INT
R
RUN
TAN
E
REM
COS
W
DRAW
SIN
Q
PLOT
VAL
J
LOAD
SQR
H
GOSUB
ABS
G
GOTO
SGN
F
FOR
DATA
D
DIM
RESTORE
S
SAVE
READ
A
NEW
IN KEY $
N
NEXT
BIN
B
BORDER
L LIST
V
CLS
L PRINT
C
CONT
EXP
X
CLEAR
LN
Z
COPY
CAPS SHIFT
M
PAUSE
K

DK

Screen Shot

PROGRAMMING SERIES

STEP-BY-STEP PROGRAMMING

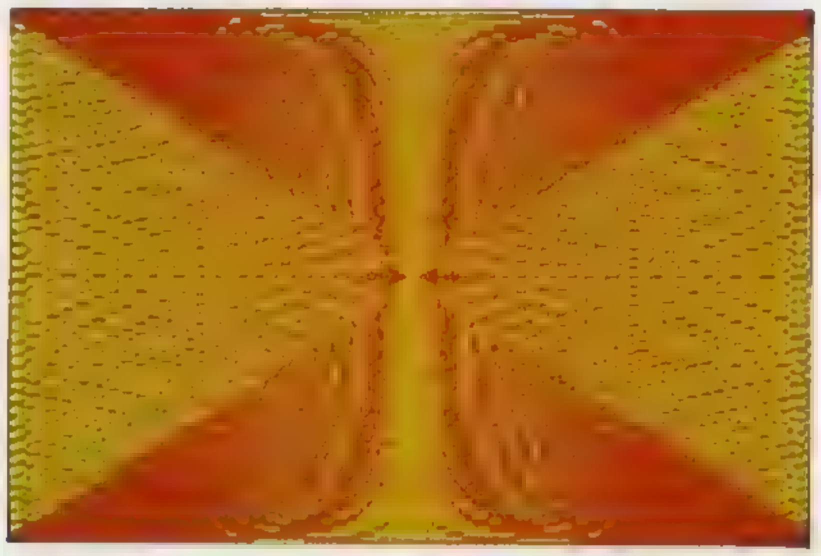
ZX SPECTRUM

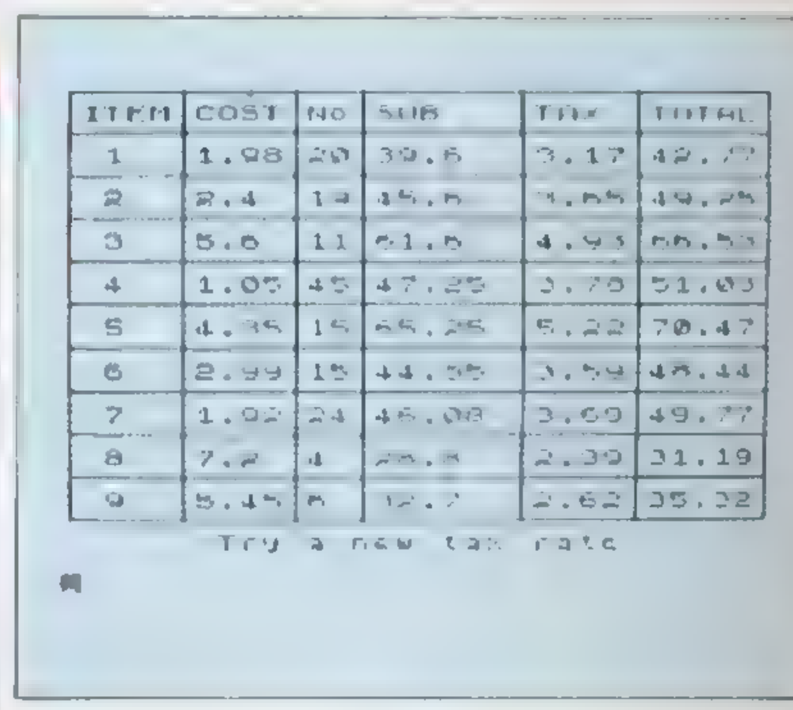
IAN GRAHAM

DORLING KINDERSLEY·LONDON

BOOK TWO

CONTENTS





The DK Screen-Shot Programming Series was conceived, edited and designed by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.

Project Editor David Burnie
Art Editor Peter Luff
Designer Hugh Schermuly
Photography Vincent Oliver
Managing Editor Alan Buckingham
Art Director Stuart Jackman

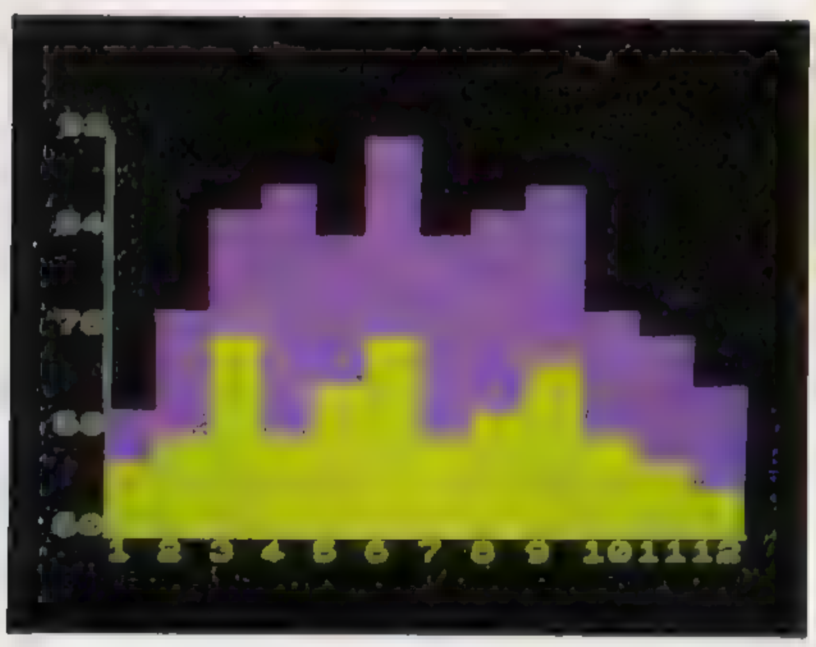
First published in Great Britain in 1984 by Dorling Kindersley Limited, 9 Henrietta Street, Covent Garden, London WC2E 8PS.
Second impression 1984

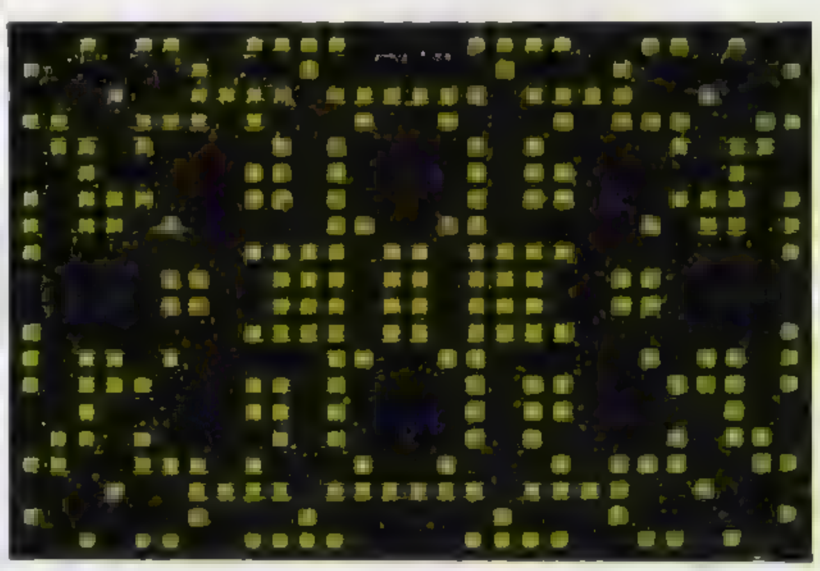
British Library Cataloguing in Publication Data

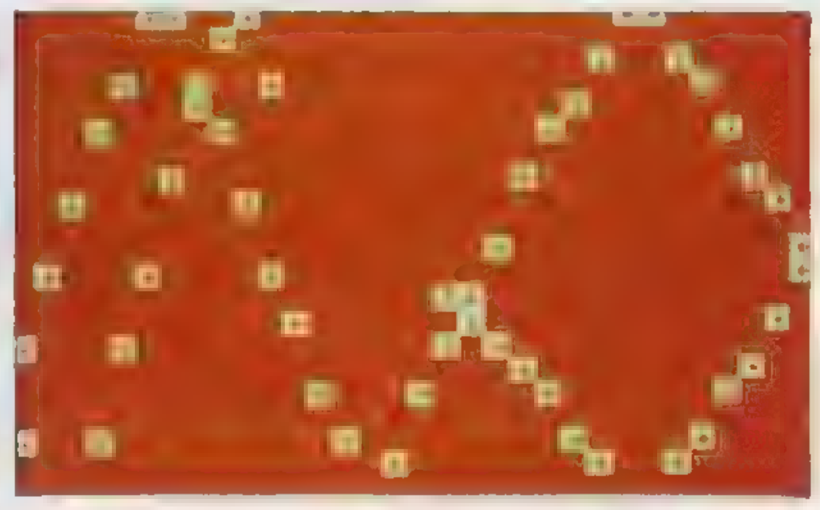
Graham, Ian, 1953–
Step-by-step programming for the ZX Spectrum. Book 2.
1. Sinclair ZX Spectrum (Computer)——Programming
I. Title
001.64'2 QA76.8.S625

ISBN 0-86318-031-0

Typesetting by The Letter Box Company (Woking) Limited, Woking, Surrey, England
Reproduction by Reprocolor Llovet S.A., Barcelona, Spain
Printed and bound in Italy by A. Mondadori, Verona







DEFINING FUNCTIONS

All computers feature a range of built-in functions, commands which can be used to transform one number into another in a specific way. Functions produce a result that can be used later in a program. SQR (SQuare Root) or INT (INTeger) are two examples of functions that are pre-programmed on the Spectrum. When you use these commands, they take a number and operate on it to produce another number.

The range of keyboard functions on the Spectrum is quite wide. But if you want to use a function that does not appear on a key, you don't have to type out a chain of instructions every time. The Spectrum allows you to program it to carry out specific sequences of calculations. These functions are "called" by the command FN (FunctioN) and defined by the command DEF FN (DEFine FunctioN).

How to write a function

In order to use a function, you must first define what it is going to do. That is done with a defining statement. For instance:

```
120 DEF FN a(x)=4*x+36
```

defines a function called "a". The number that a function operates on is known as its argument. In this case the argument of the function is x. The function takes whatever value of x it is given, multiplies it by 4, and then adds 36. If in a program you want to put the number 10 into the function, you would do so by using the keyword FN like this:

```
200 PRINT FN a(10)
```

This would PRINT the value of the function when 10 is substituted for x – which is 4*10+36, or 76.

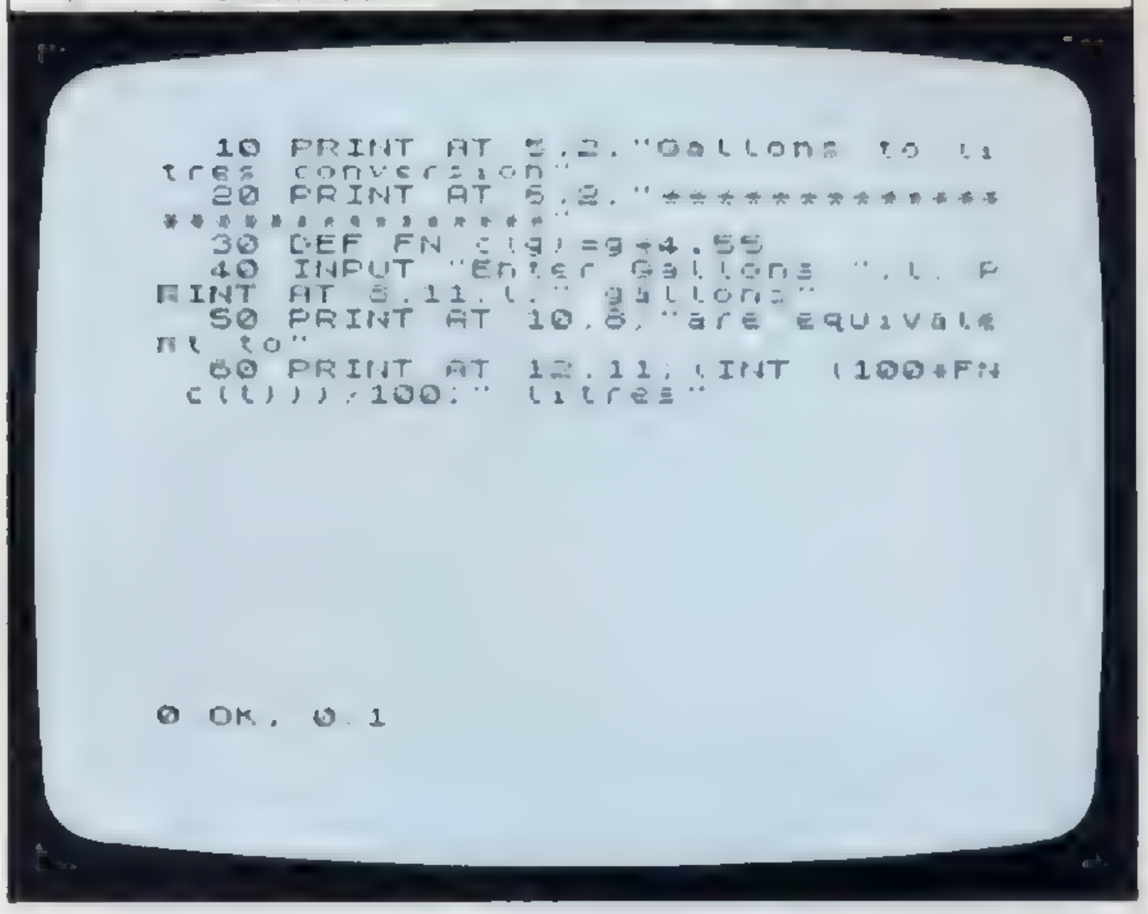
DEF FN is obtained by pressing CAPS SHIFT together with SYMBOL SHIFT, followed by SYMBOL SHIFT again to get the extended mode cursor, together with key 1. The same procedure, but using key 2 instead of key 1, produces FN.

Once a function has been defined in a program, you can use it and its argument just like any other number or numeric variable. For example, you can add, subtract, divide and multiply functions and their arguments together, and even make functions work on numbers that are themselves produced by functions. Unless you are doing mathematical research you are unlikely to get this far, but for more straightforward tasks functions are easy to use and helpful in making programs simpler.

Why use a function?

The following program shows a simple way in which you can put functions to work in a program to produce a numerical result which is then PRINTed:

FUNCTION CONVERSION PROGRAM



The program carries out a conversion. The function that actually does the converting is defined in line 30. Line 40 waits for you to type in a number of gallons, which is then converted into the equivalent number of litres by FN c(l) in line 60. Multiplying and dividing by 100 may look odd, but it's just a way of producing an answer to two places of decimals with INT. Using INT on its own removes all decimals, and so reduces accuracy.

Going to the trouble of using a function here might seem a bit unnecessary, and in fact it's unlikely that you would use FN in such a simple program. But imagine what would happen if you wanted to do the calculation a number of times at different places in the same program, and with different numbers. It is then that the user-defined function really comes into its own. When the function is long and complicated, defining it just once enables you make calculation lines much simpler to write and check. FN is very much like a one-

command subroutine that deals only with numbers.

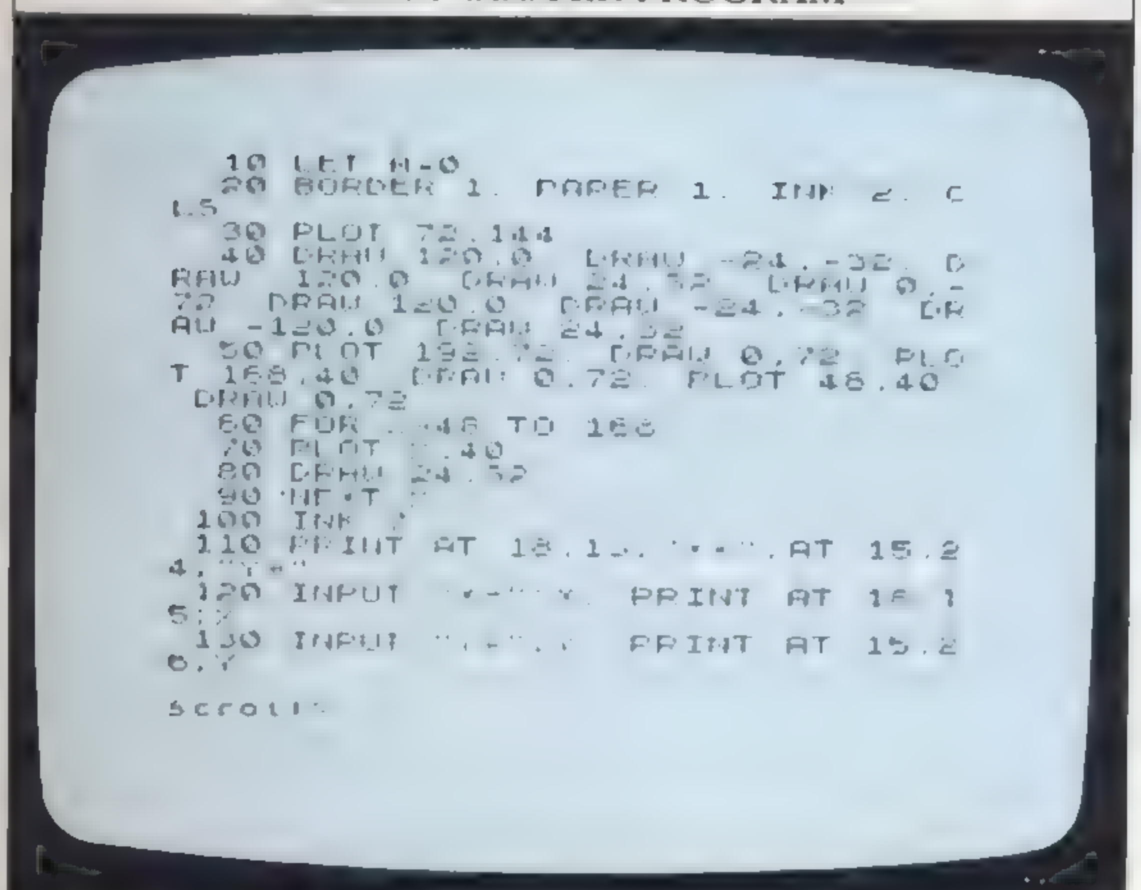
Because an expression containing FN actually represents a number, you can use it to replace any kind of complex calculation. When you write your own functions, you are in effect giving the computer functions that its resident programming language (BASIC) doesn't already have – extending the capabilities of the language.

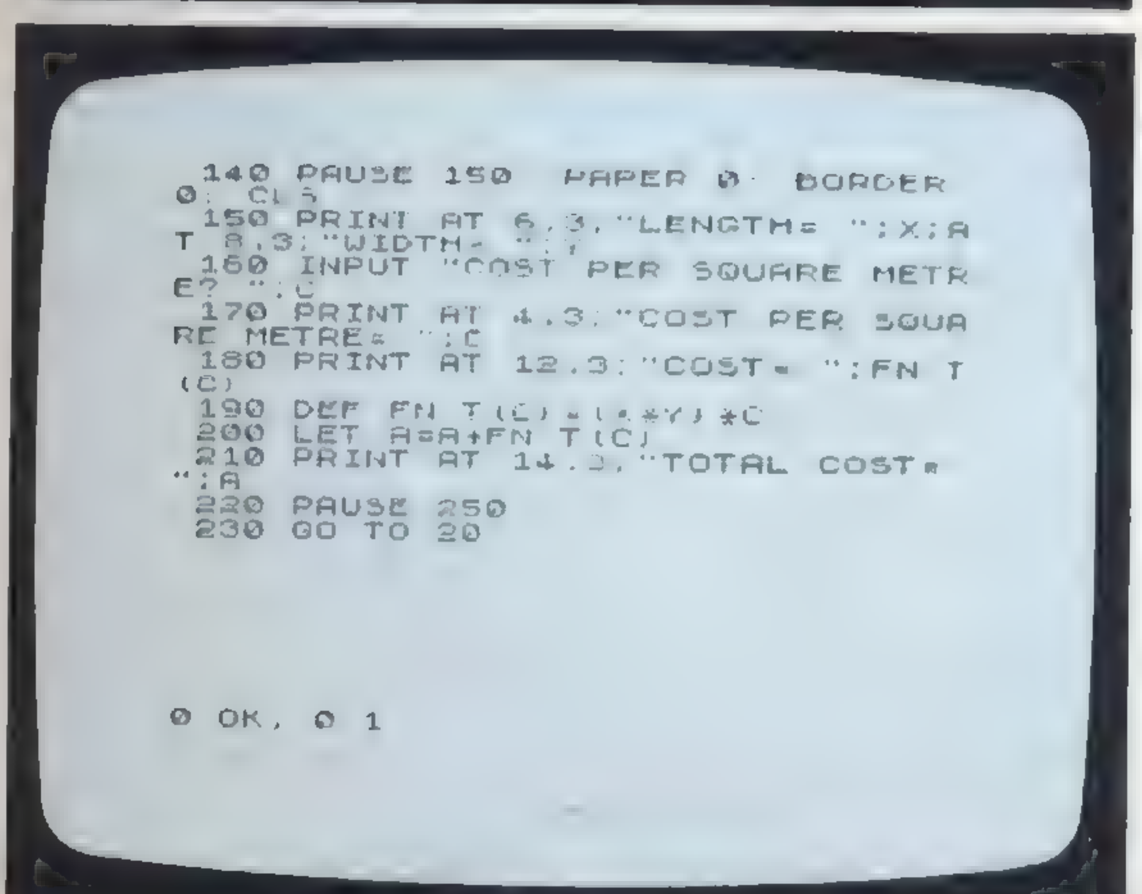
Calculation sequences with functions

Imagine that you want to calculate the cost of something that is sold by area – perhaps carpets to cover the floor of a house. You would need to multiply the length and width of each room to get its area, and then multiply that by the cost of the floor covering per square metre. If you called the length and width X and Y, and the cost per unit area C, then the cost per room would be worked out by (X*Y)*C.

In the next program, the cost for each room is calculated by a function. It is defined in line 190, and used in lines 180 and 200:

CARPET COSTER PROGRAM

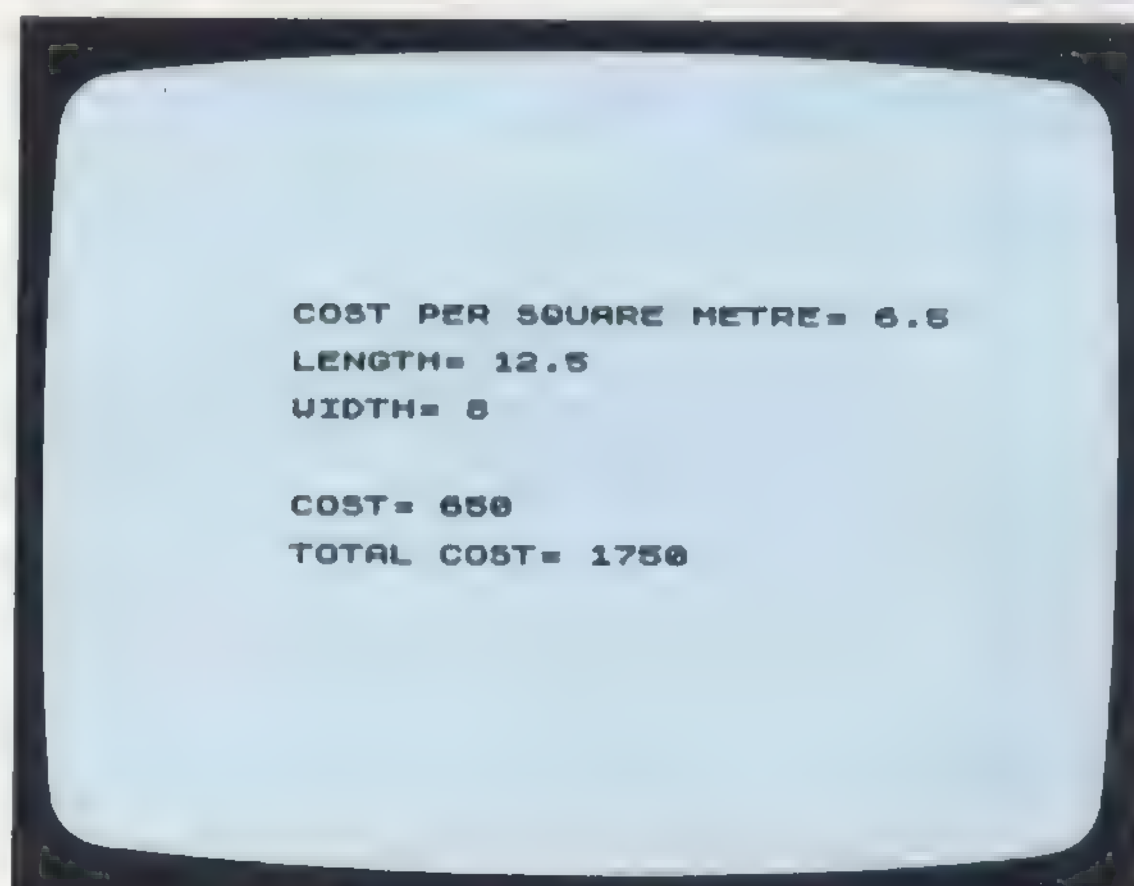




Lines 20 to 130 set up a graphics display which produces an outline of the room, and then wait for you to ENTER values for length and width. Once you have done this, the program INPUTs the values and then clears the screen.

The next display asks you for the cost per square metre of the floor covering. Once you have keyed this in, the program tells you what it would cost to cover the room. As well as using the function T to produce this, it uses it in line 200 to update a running total. If you keep keying in values every time the program returns to the first display, you will find that the TOTAL COST line in the second display will be updated to show the running total of all the costs calculated:

CARPET COSTER DISPLAYS



You can define a function at any point in a program, although if the DEF FN and FN lines are far apart in a long program, this will increase the program's RUNning time. In the carpet coster program, using a function makes lines 180 and 200 less complicated than they otherwise would be. On page 33 you will see how using a function can make graphics far easier as well.

EXTENDING DECISIONS

The BASIC keywords IF and THEN let a program operate in one way until the condition specified by the IF statement is encountered. When this happens, the program is then triggered to follow another course of action. But the capabilities of IF … THEN do not stop at making a straightforward "yes" or "no" decision. By combining IF … THEN with the keywords AND and OR you can make it tackle much more complicated situations. Because BASIC is designed to reflect how words are used in ordinary language, you can use IF … THEN just as you would when describing a set of conditions to someone. Here is a program which shows how you can take IF … THEN decision-making to a more advanced level:

BOUNCING PROGRAM

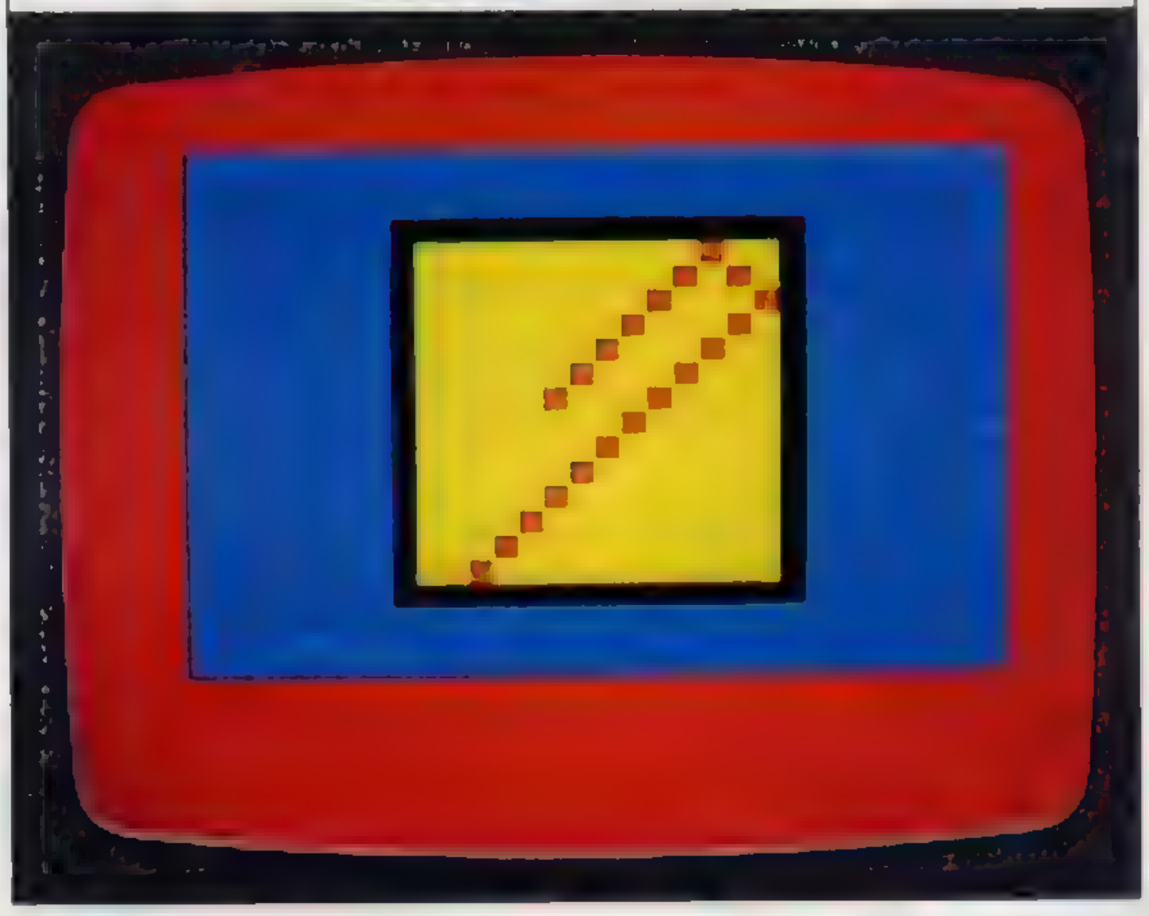
```
  10 BORDER 2: PAPER 1: INK 0: C
LS
  20 FOR r=3 TO 18
  30 PRINT AT r,8;"■"
  40 PRINT AT r,23;"■"
  50 PRINT AT 3,r+5;"■"
  60 PRINT AT 18,r+5;"■"
  70 NEXT r
  80 PAPER 6: INK 2
  90 FOR r=4 TO 17
 100 FOR c=9 TO 22
 110 PRINT AT r,c;" "
 120 NEXT c
 130 NEXT r
 140 LET r=5+INT (RND*12)
 150 LET c=10+INT (RND*12)
 160 LET a=1: LET b=1
 170 PRINT AT r,c;" "
 180 LET r=r+a
 190 LET c=c+b
 200 IF r=4 OR r=17 THEN LET a=-
a

scroll?
```

```
 210 IF c=9 OR c=22 THEN LET b=-
b
 220 PRINT AT r,c;"■"
 230 BEEP 0.1,3*r
 240 GO TO 170



0 OK, 0:1
```

Lines 10 to 130 simply set up the display – a black outline box in the middle of the screen. Lines 140 and 150 produce a random starting position for a "ball" (a graphics square) inside the box. To make the ball appear to move, line 170 erases the image at r,c and lines 180 and 190 produce new co-ordinates where line 220 will rePRINT the ball.

Before this happens, lines 200 and 210 check whether the ball has reached any of the box's walls. They examine the ball's position to see if the row number is one below the box lid or one above the box bottom, or if its column number is one more than the left side or one less than the right side. If any of these conditions is satisfied, lines 200 or 210 reverse the ball's vertical or horizontal motion, whichever is necessary:

BOUNCING DISPLAY



Adding decisions together

You can now move on a stage further from the previous program to see how a second option can be incorporated in an IF … THEN statement. The next program features two balls moving independently:

DOUBLE BOUNCING PROGRAM (1)

```
  10 BORDER 2: PAPER 1: INK 0: C
LS
  20 FOR r=3 TO 18
  30 PRINT AT r,8;"■";AT r,23;"■
"
  40 PRINT AT 3,r+5;"■";AT 18,r+
5;"■"
  50 NEXT r
  60 PAPER 6: INK 2
  70 FOR r=4 TO 17
  80 FOR c=9 TO 22
  90 PRINT AT r,c;" "
 100 NEXT c
 110 NEXT r
 120 LET r=5+INT (RND*12)
 130 LET s=5+INT (RND*12)
 140 LET c=10+INT (RND*12)
 150 LET d=10+INT (RND*12)
 160 LET a=1: LET b=1
 170 LET k=-1.3: LET l=1
 180 PRINT AT r,c;" "
 190 PRINT AT s,d;" "

scroll?
```

This program is very similar to the first one, except that now there are two lines that PRINT a graphics square at changing row and column numbers. As before, each of the squares starts at a random co-ordinate which is defined in lines 120 to 150. The squares are then animated by lines 160 to 300:

DOUBLE BOUNCING PROGRAM (2)

```
200 LET r=r+a: LET c=c+b
210 LET s=s+k: LET d=d+l
220 IF r=4 OR r=17 THEN LET a=-a
230 IF s<5 OR s>16 THEN LET k=-k
240 IF c=9 OR c=22 THEN LET b=-b
250 IF d=9 OR d=22 THEN LET l=-l
260 PRINT AT r,c;"■"
270 PRINT AT s,d;"■"
280 IF r>=INT (s+0.5)-1 AND r<=INT (s+0.5)+1 AND c>=d-1 AND c<=d+1 THEN BEEP 0.5,-10: GO TO 10
290 BEEP 0.02,3*r: BEEP 0.02,3*s
300 GO TO 160

0 OK, 0:1
```

The second ball is made to set off in a different direction from the first and at a slightly faster vertical speed, so that the two balls have a greater chance of meeting. Otherwise, they would just follow each other around the box in the same tracks. Line 280 is the one in which the computer makes a multiple decision about the position of both of the squares. Without this line, when the squares met, they would just carry on through each other as if nothing had happened. This isn't a very convincing simulation of what would really occur, so line 280 decides whether the squares are close enough together to have collided. The line includes an IF ... THEN decision with three ANDs to see if r and s are sufficiently close together, and then if c and d are also within the same limits. It does this by taking r, for example, and then deciding whether it is greater or equal to INT(s+0.5)−1 and simultaneously smaller or equal to INT(s+0.5)+1.

If all these conditions are met, then it means that the two balls are either occupying the same square or are on adjacent squares, in which case they can be assumed to have collided. A BEEP is sounded and the whole process starts again.

IF ... THEN in games programming

Decision-making by this method is very useful if you want to know whether or not two characters are occupying the same screen location. This is often used in program where a character is "shot down" by another. The next program shows how you can incorporate the technique in a simple game which has a number of characters moving simultaneously:

FIRE-BASE PROGRAM

```
10 BORDER 1: PAPER 0: INK 6: CLS
20 PRINT AT 19,15;"▚ ▞";AT 20,16;"■";AT 21,14;"▄▀▀▄"
30 LET c=0: LET d=7: LET r=18
40 PRINT AT 3,c;" ";AT 7,d;" "
50 LET c=c+1: LET d=d+1
60 IF c>31 THEN LET c=0
70 IF d>31 THEN LET d=0
80 PRINT INK 7;AT 3,c;">";AT 7,d;">"
90 PRINT AT r,16;" "
100 IF r=0 THEN LET r=19
110 LET r=r-1
120 PRINT AT r,16;"↑": BEEP 0.02,800/(3*r+15)
130 IF r=3 AND c=16 OR r=7 AND d=16 THEN BEEP 0.5,r↑2: GO TO 10
140 GO TO 40

0 OK, 0:1
```

The program PRINTs a fire-base at the bottom of the screen. It fires upward arrows at two horizontal arrows that repeatedly fly across the screen. Line 130 checks whether the screen co-ordinates of the upward arrow are the same as those of either of the horizontal arrows. If they are, the program jumps right back to line 10 and begins again. If not, it jumps back to line 40 and moves all the characters on one space. Line 100 checks whether the upward arrow has reached the top of the screen. If it has, a new arrow is launched from just above the fire-base. Lines 60 and 70 check whether either of the horizontal arrows have reached the right edge of the screen. If either has, its column co-ordinate is reset to zero.

When you RUN the program, you should find that the fire-base's arrow finds its target on the horizontal arrows' seventh pass across the screen. This happens because the program is working with fixed figures. If you use RND instead, the results become unpredictable and will change with each RUN:

FIRE-BASE DISPLAY

CHANGING STEP IN GRAPHICS

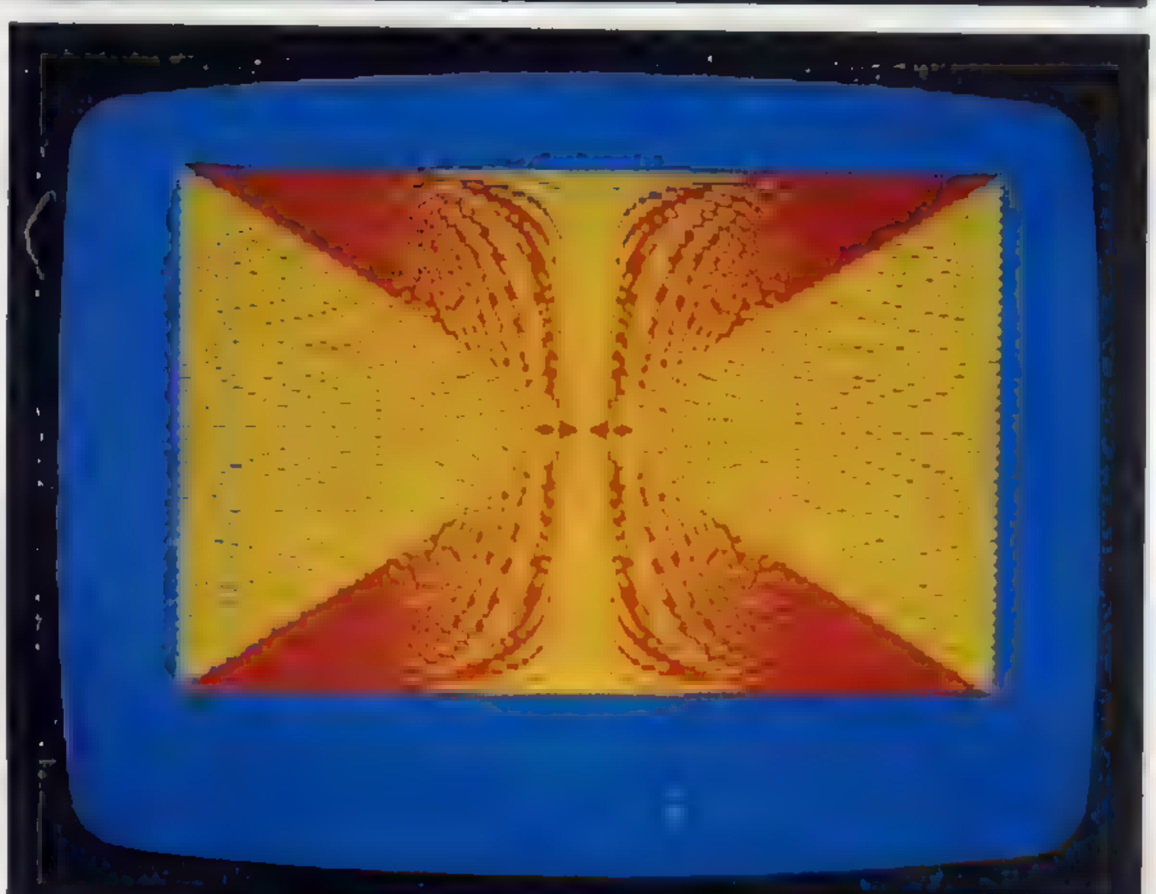
FOR ... NEXT loops are very useful for repeating a sequence of program statements a predictable number of times. If you want to increase the value of a variable by more than 1 on every cycle through a loop, you could replace FOR ... NEXT by a GOTO statement preceded by N=N+2 or whatever change you want to make to N. However, if you have already started writing a program and then find that the standard FOR ... NEXT loop is unsuitable, it is sometimes awkward and time-consuming to substitute a completely new programming technique like this. Fortunately, there is a much more straightforward way of jumping forwards or even backwards in a loop.

How to change jump sizes with STEP

The BASIC keyword that deals with this is STEP, which you may already have come across in Book 1. Here is a program which uses STEP with graphics:

GRAPHICS WITH STEP

```
  10 BORDER 1: PAPER 6: INK 2: C
LS
  20 FOR y=0 TO 175 STEP 4
  30 PLOT 0,0
  40 BEEP 0.1,y/3
  50 DRAW 255,y
  60 PLOT 255,0
  70 DRAW -255,y
  80 PLOT 0,175
  90 DRAW 255,-y
 100 PLOT 255,175
 110 DRAW -255,-y
 120 NEXT y

0 OK, 0:1
```



The graphics cursor is moved to each of the four corners of the screen in turn. A line is DRAWn from each corner to a point on the opposite side of the screen. The line separation is determined by the STEP size in line 20. STEP 4 gives the best results. This makes the values of y 0, 4, 8, 12, 16, and so on, instead of y increasing by 1 in every circuit of the loop.

Using STEP to DRAW grids

One display that can be produced very easily using STEP is a games grid. You could make a grid just by DRAWing a series of criss-crossing lines against a contrasting background. But instead of specifying each line, you can use loops with STEP to do most of the work for you:

GAMES GRID PROGRAM

```
  10 BORDER 6: PAPER 2: INK 7: C
LS
  20 FOR y=25 TO 151 STEP 21
  30 PLOT 26,y
  40 DRAW 204,0
  50 NEXT y
  60 FOR x=26 TO 230 STEP 34
  70 PLOT x,25
  80 DRAW 0,126
  90 NEXT x

0 OK, 0:1
```

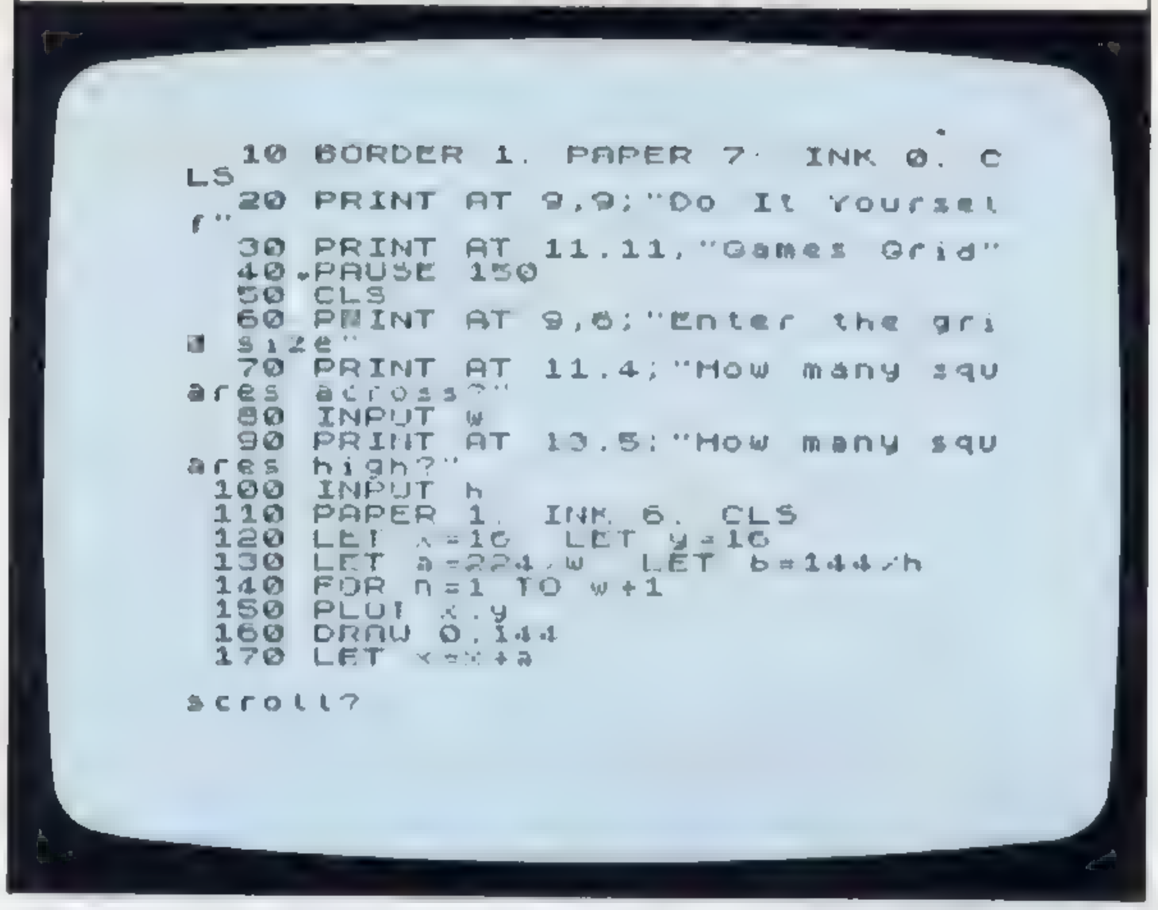
Lines 20 to 50 DRAW a series of lines across the screen at vertical intervals of 21. Then lines 60 to 90 DRAW lines down the screen with a separation of 34. (The co-ordinates and line separations given here may be easier to follow if you refer to the graphics layout grid on page 59.)

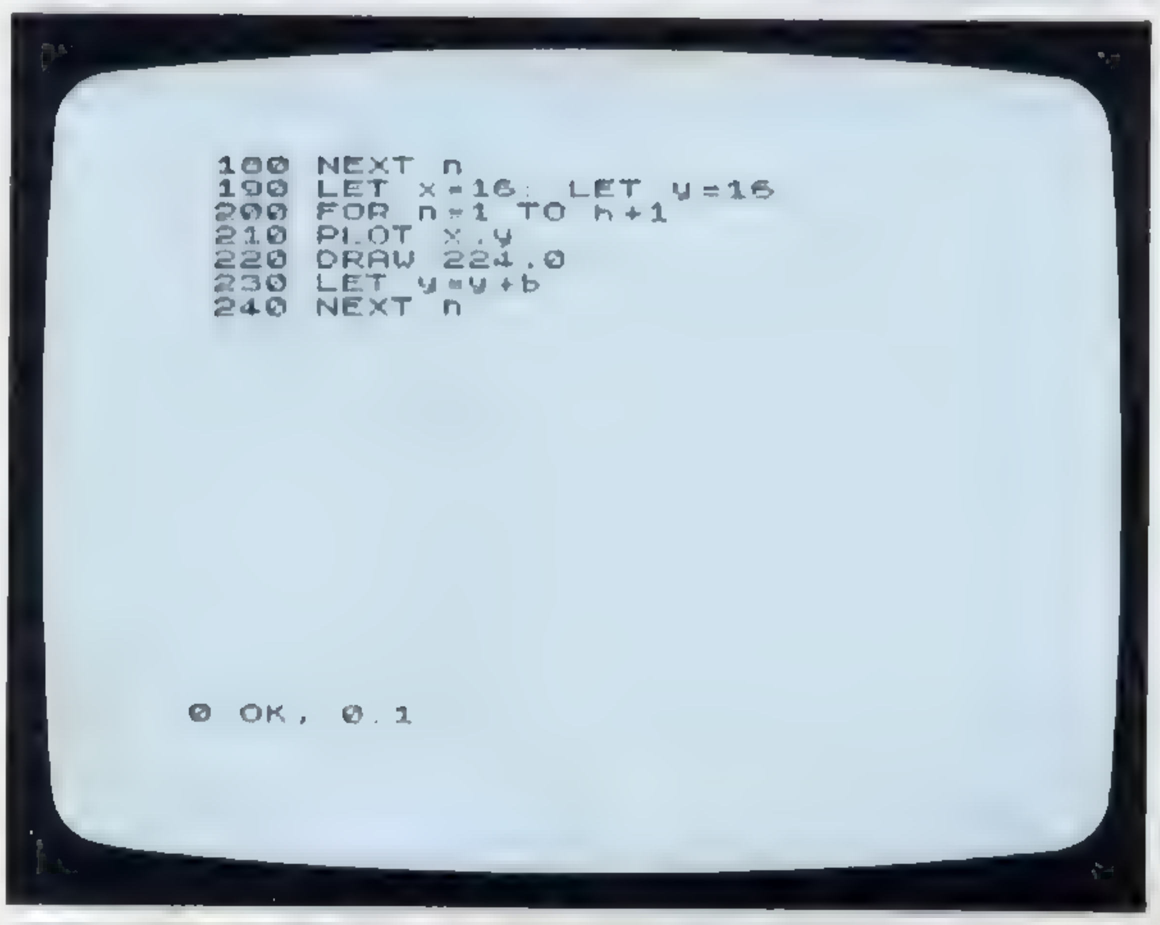
Next is a program that will produce a games grid with as many squares as you like. The program asks you what sort of grid you want (how many squares across and down) and then feeds those numbers into a subroutine describing a generalized grid pattern.

The program is divided into three sections. First the INPUT section (lines 10 to 100) asks you for the information necessary for the second half of the program. This takes the numbers you supply and uses them to calculate the separations of the lines DRAWn in the final display section (lines 140 to 240). It does this with two variables; w, which represents the number of squares across, is divided into the screen's display panel width in line 130. In the same line h, the number of

squares from top to bottom, is divided into the height. With this program you can produce an almost endless variety of grids containing either squares or rectangles, depending on the figures keyed in:

VARIABLE GRID PROGRAM



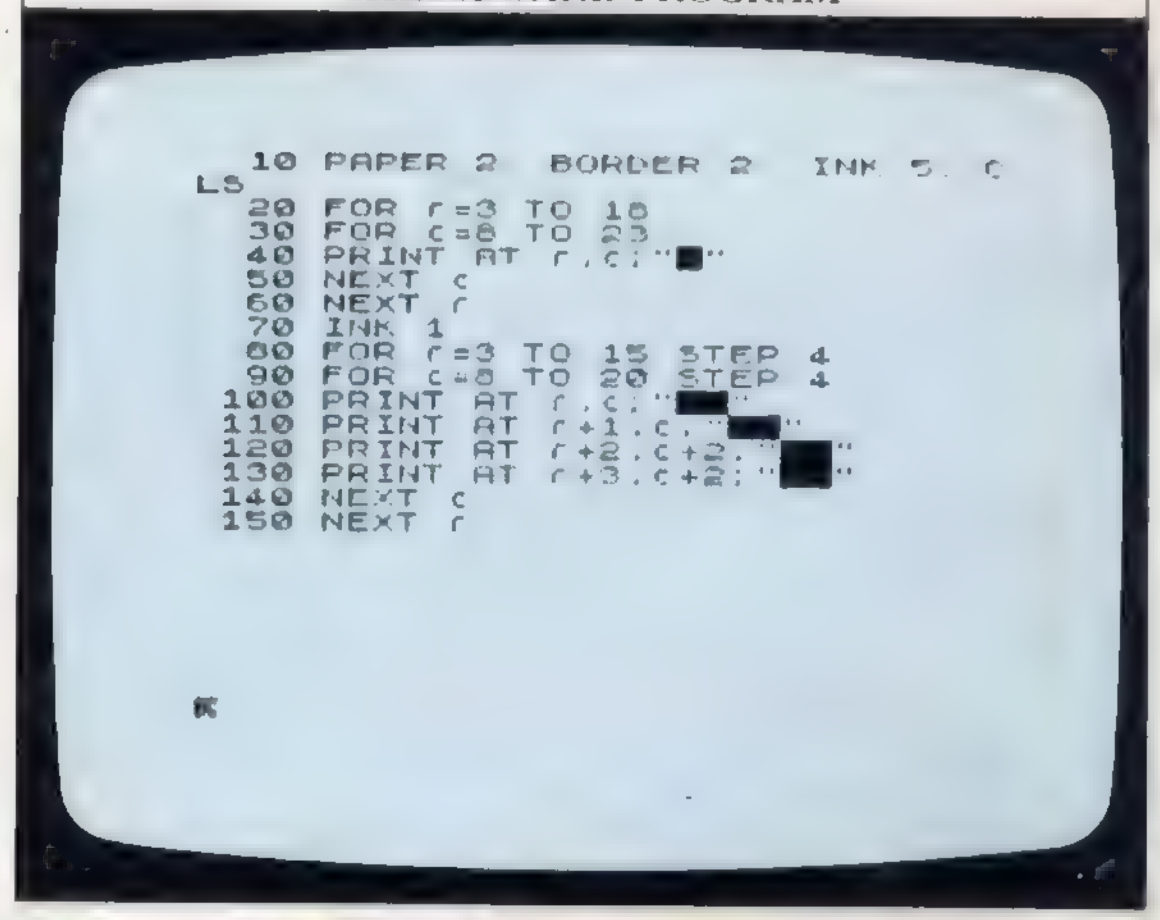


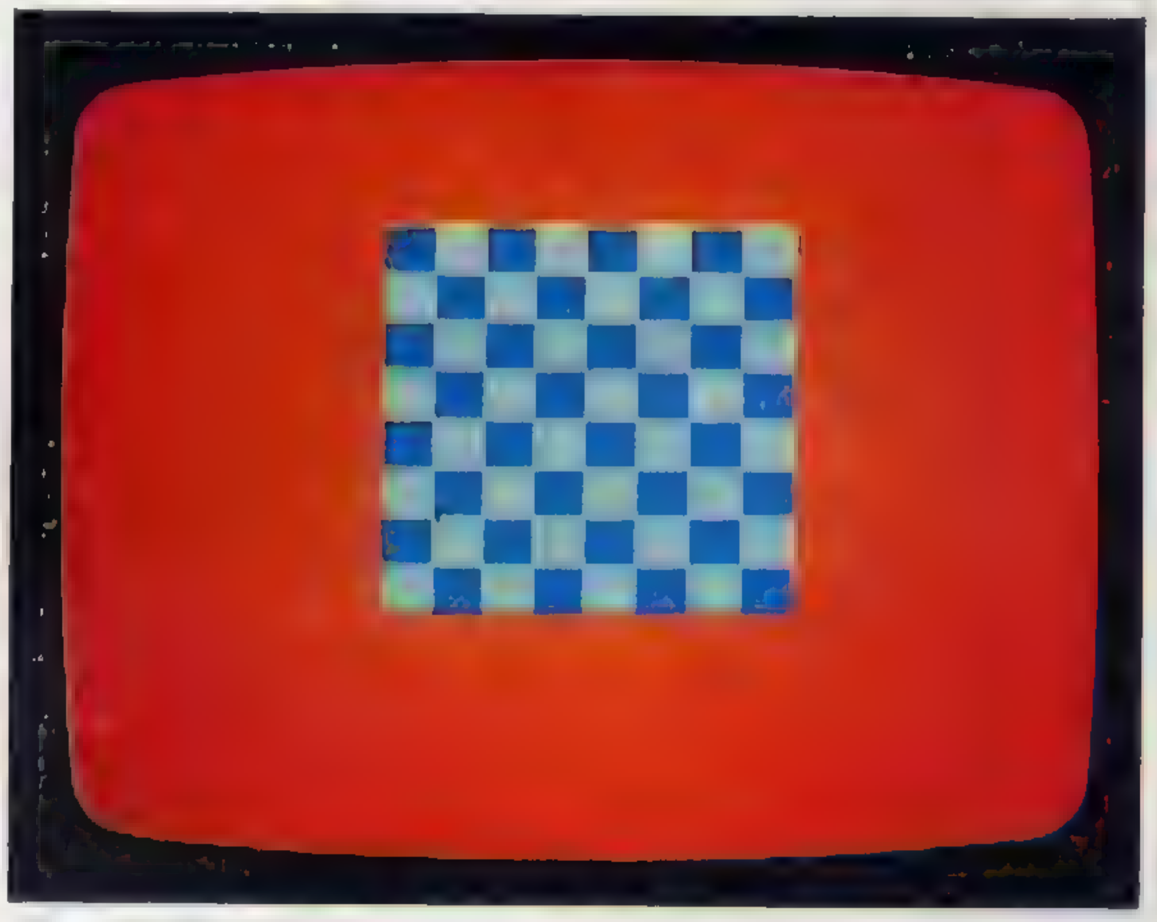
You can use this type of program as the basis for a chart, and then PRINT figures inside each square or rectangle. By using a technique which you will encounter on pages 22–23, you can then make the figures within the chart respond to different INPUTs.

How to PRINT a chessboard

To program the computer to PRINT a chessboard, you can PRINT alternate white and black squares, working your way down the board from top to bottom. It is then a simple matter then to add some colour to the display by introducing BORDER, PAPER and INK:

CHESSBOARD PROGRAM





Here PRINT AT is used to build up rows of dark blue squares over a cyan background. The program selects a blue foreground colour (INK 1) and sets the row co-ordinate to the top line of the board. Having PRINTed one blue square, it skips the second one, leaving it cyan, and PRINTs a blue square in the third position. The column STEP size is set to 4 to deal with this. As each loop draws two rows of squares, the row STEP size of 4 skips every other row.

INFORMATION FROM THE KEYBOARD

To key in new information while a program is RUNning you have – until now – used the command INPUT. With INPUT you must press ENTER after keying the information in. This technique has its disadvantages. Even when you know that it's necessary to use ENTER, you can occasionally forget, and using two keys in sequence also slows programs down. It is much more useful if the computer responds without waiting for you to press ENTER, just as arcade machines respond every time you press a control button. To make the Spectrum do this, you can use the keyword INKEY$.

ASCII codes

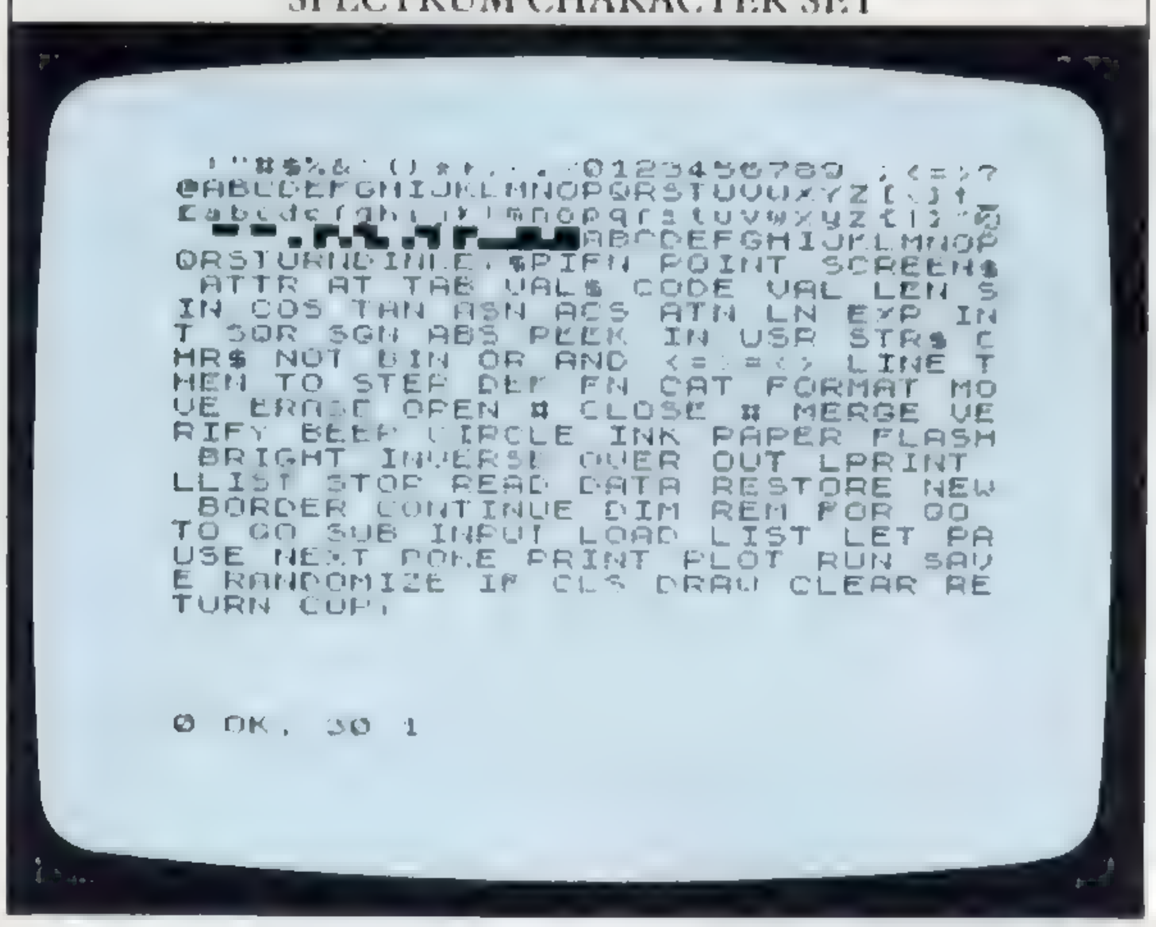
Keywords, numbers, letters, punctuation marks and mathematical symbols – in fact, all the symbols that the computer recognizes – are stored inside the computer as numbers from 0 to 255. Most computers, including the Spectrum, store these numbers according to a code called the American Standard Code for Information Interchange (ASCII). So, every computer that uses the ASCII standard stores a letter A as the decimal number 65, a "+" sign as 43, and the decimal number 1 as 49. (A complete table of ASCII codes is shown on page 60.)

You can see all the characters that the Spectrum uses by keying in a very simple program:

```
10 FOR n=33 TO 255
20 PRINT CHR$ n;
30 NEXT n
```

This makes the computer take each ASCII code in turn and then PRINT the character that the code represents, missing out the initial "blanks" and control codes:

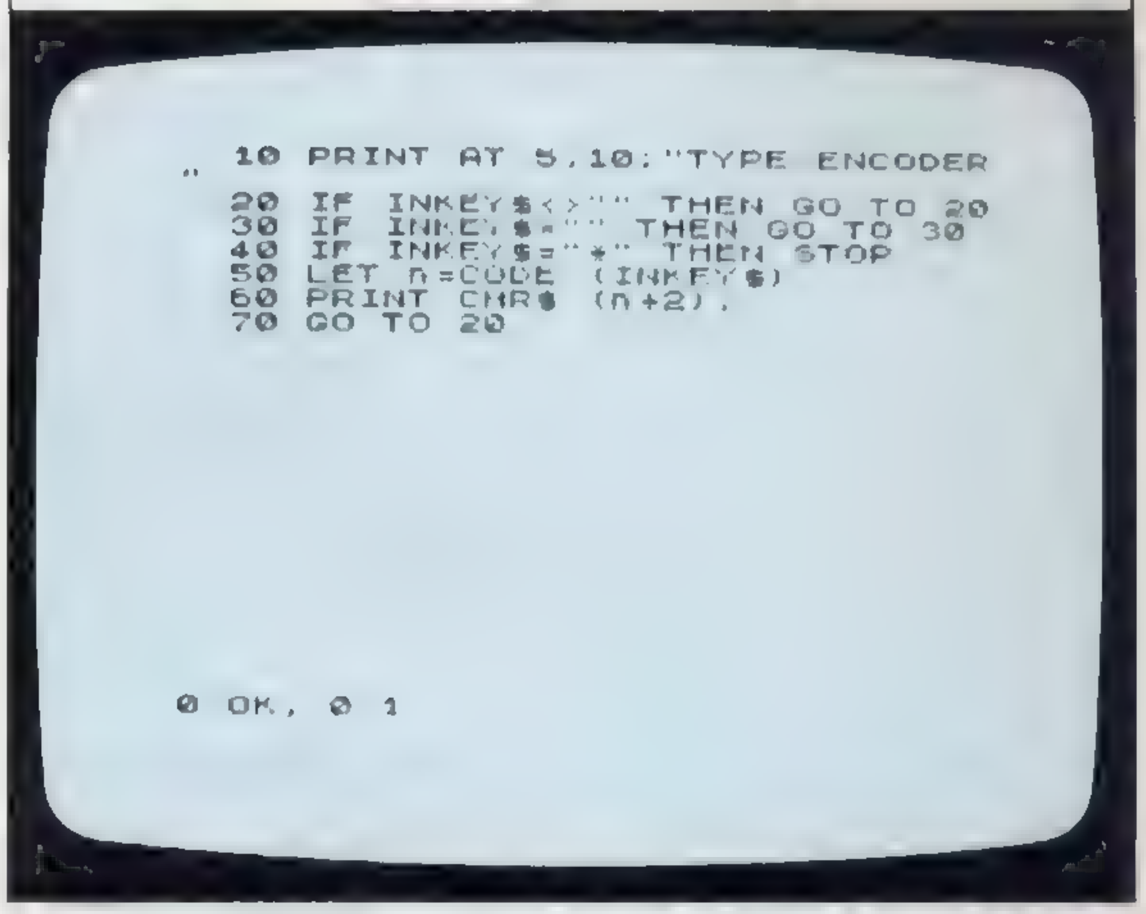
SPECTRUM CHARACTER SET



You can see from this that some codes represent just one letter or number, whereas others represent complete BASIC keywords.

The next program uses INKEY$ to respond to keys as they are pressed, and then uses ASCII numbers to PRINT in a simple code:

ENCODER PROGRAM



Line 10 PRINTs the title and lines 20 and 30 use INKEY$ to wait for you to release a key before continuing. Line 20 may puzzle you. Change it to:

```
20 REM IF INKEY$<>"" THEN GOTO 20
```

so that it doesn't work. You should find it impossible to press a key and get a single character on the screen. Unlike INPUT, INKEY$ doesn't wait for you – it checks if any key has been pressed and then moves on to the next line. So line 20 keeps on being carried out until you release a key, then line 30 keeps on RUNning until you press the next key.

Line 40 then tests the keyed character to see if it is an asterisk. If it is, the program ends. If not, line 50 converts your character into its ASCII code. Line 60 adds 2 to the code for the key you have pressed and then PRINTs the equivalent character. So, the program appears to PRINT nonsense when you key something in. There's no point in producing a coding program if you can't crack the code. You can quickly turn the encoder program into a decoder by inserting:

```
10 PRINT AT 5,10;"TYPE DECODER"
60 PRINT CHR$(n-2);
```

To see if it works try keying in the following when the program is RUNning:

ngctp"vq"rtqitco"ykvj"vjg"FM"uetggp"ujqv"ugtkgu

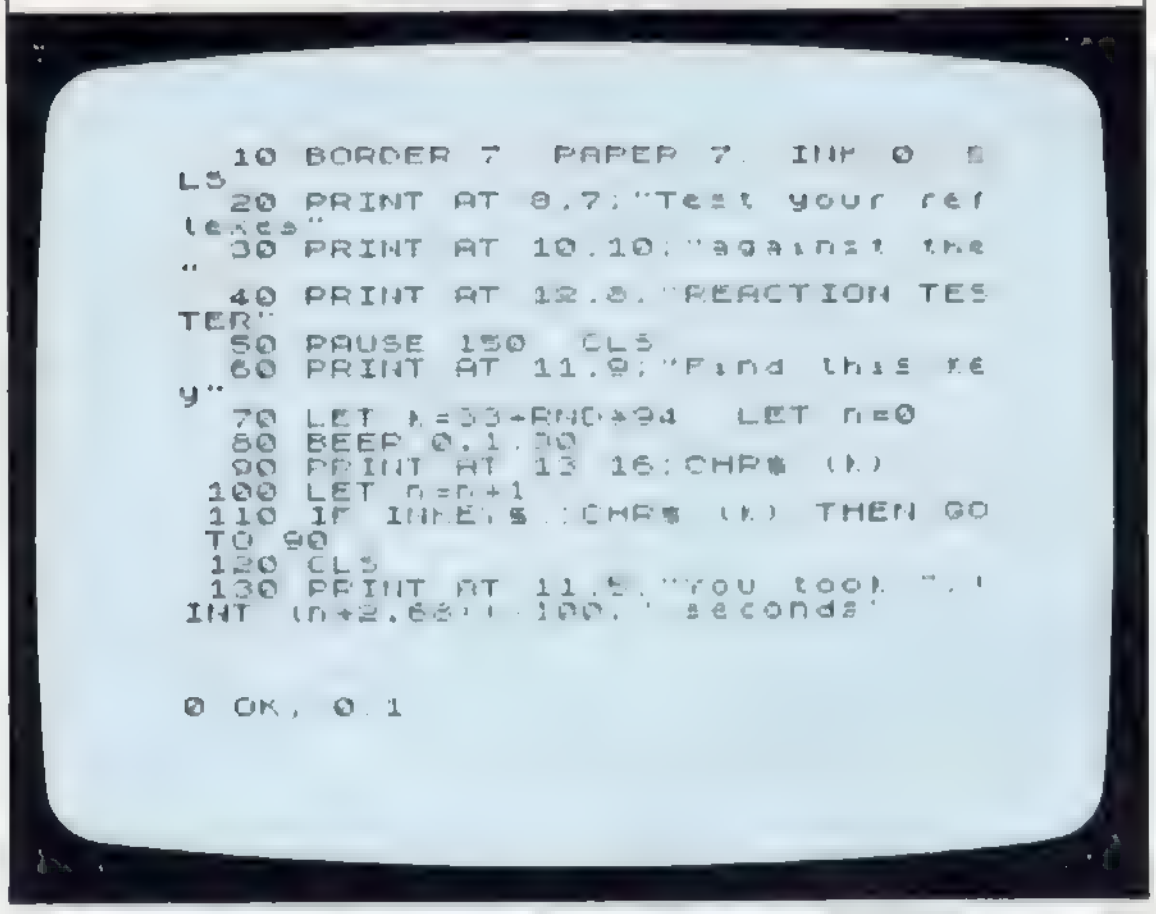
Testing your reactions with INKEY$

You can use INKEY$'s ability to check the keyboard in combination with ASCII codes to write a program that

enables you to time the speed of your reactions.

This program produces a random symbol on the screen, and then times how long it takes you to press the symbol it has selected:

REACTION TESTER



Lines 10 to 50 PRINT the title frame and PAUSE for 3 seconds. Then the game begins. Line 70 sets k equal to a randomly generated number between 33 and 127, the ASCII code numbers for the Spectrum's letters and symbols (this excludes the graphics symbols and keywords). Line 70 also sets n equal to zero – n is used later to work out the time taken to press the correct key. Line 80 makes a sound to signal the beginning of the timing period. Line 90 PRINTs the character equivalent to the random code found by line 70. Line 110 checks whether you have pressed the correct key. If not, it cycles back to line 90, and on each loop, n is increased by 1.

If you do press the correct key, the time elapsed since the BEEP is calculated by (INT(n*2.68))/100 seconds. If you want to know why this is the line to use, try inserting the following line:

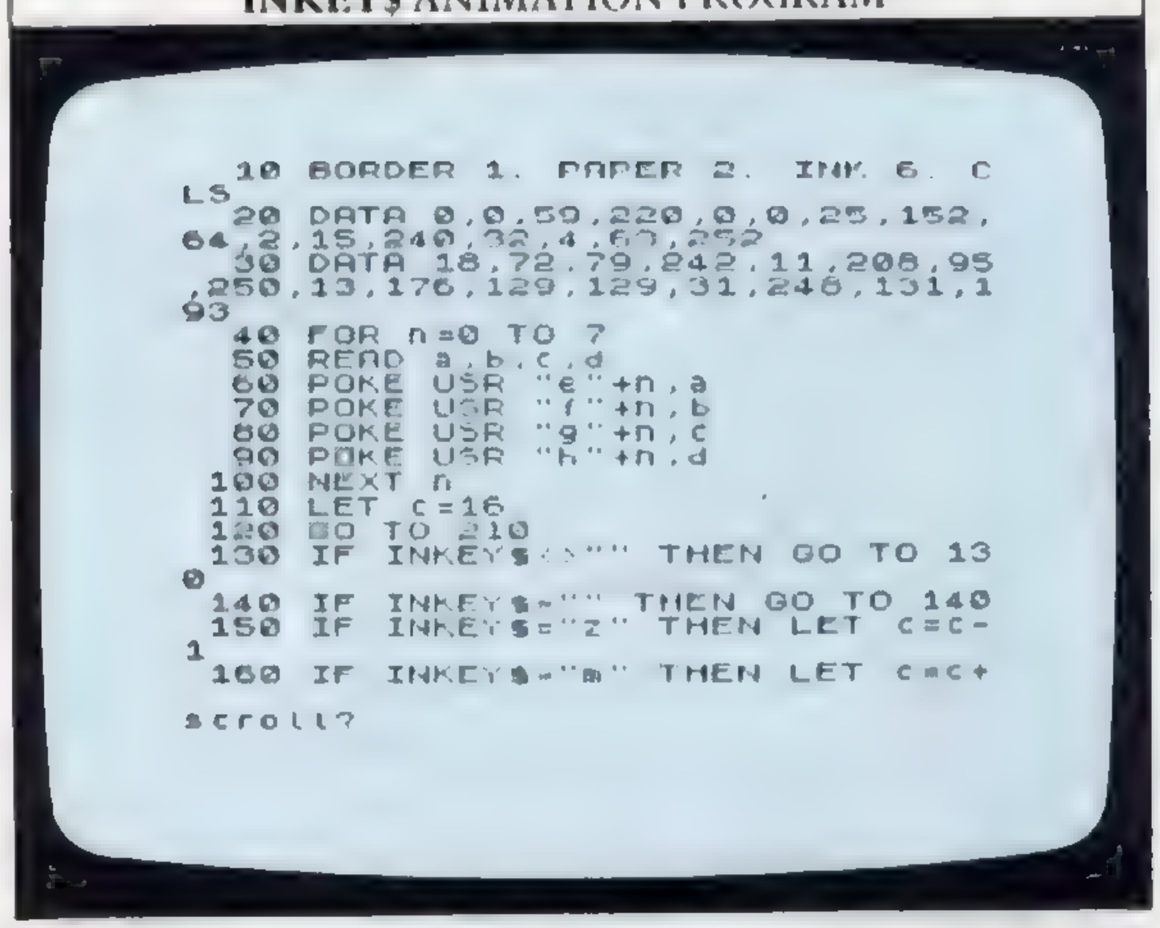
```
105 IF n=1000 THEN BEEP 0.3,20
```

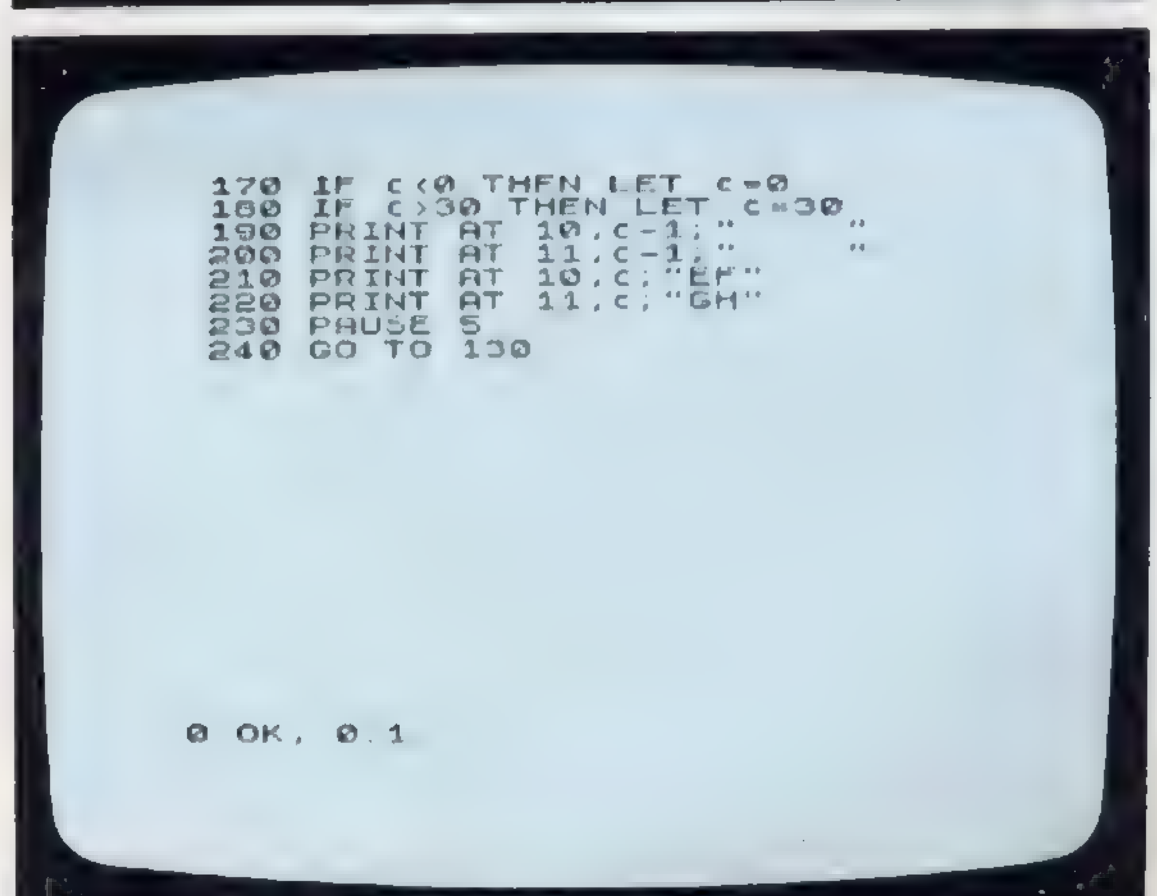
in the program. This lets you time 1000 loops with your watch. They should take roughly 26.85 seconds, so each loop takes 0.02685 seconds. To find the elapsed time, multiply n by 0.02685. To limit the time PRINTed by line 130 to two decimal places, multiply (n*0.02685) by 100 (=n*2.685), take its integer value (to get rid of all figures after the decimal point) and finally divide the result by 100.

Using INKEY$ to control movement

INKEY$'s quick responses make it ideal for controlling movement on the screen. In the following program, instead of going through a predetermined series of movements, the computer waits until either the Z or M keys are pressed and then moves the character left or right respectively. Lines 170 and 180 stop the character being moved beyond either side of the screen:

INKEY$ ANIMATION PROGRAM





If you RUN the program, you will find that the character responds only to keys Z and M.

RANDOM NUMBERS AND ODDS

The Spectrum has two separate keywords which both deal with random numbers – RND and RANDOMIZE. As you have already seen, RND is used to generate random numbers. A line like this:

```
80 LET a=RND
```

will produce a random number between 0 and 0.99999999.

Actually that's not quite true. The Spectrum has a sequence of 65,536 different numbers between 0 and 0.99999999 stored in its memory. They are all mixed up, so that there is no obvious pattern. Because of this, RND is said to be a "pseudo-random" function. There is a sequence behind the numbers, but for most purposes the numbers can be taken as completely random. Although a number between 0 and 0.99999999 isn't much use, a line like the following produces whole (integer) random numbers in other ranges:

```
100 n=1+INT(2*RND)
```

Here the line produces a 1 or 2, for a coin toss program perhaps. 2*RND produces a number between zero and 1.99999998. INT rounds that down to the next lowest integer (0 or 1), and 1 is added to this to produce a 1 or 2. In the same way:

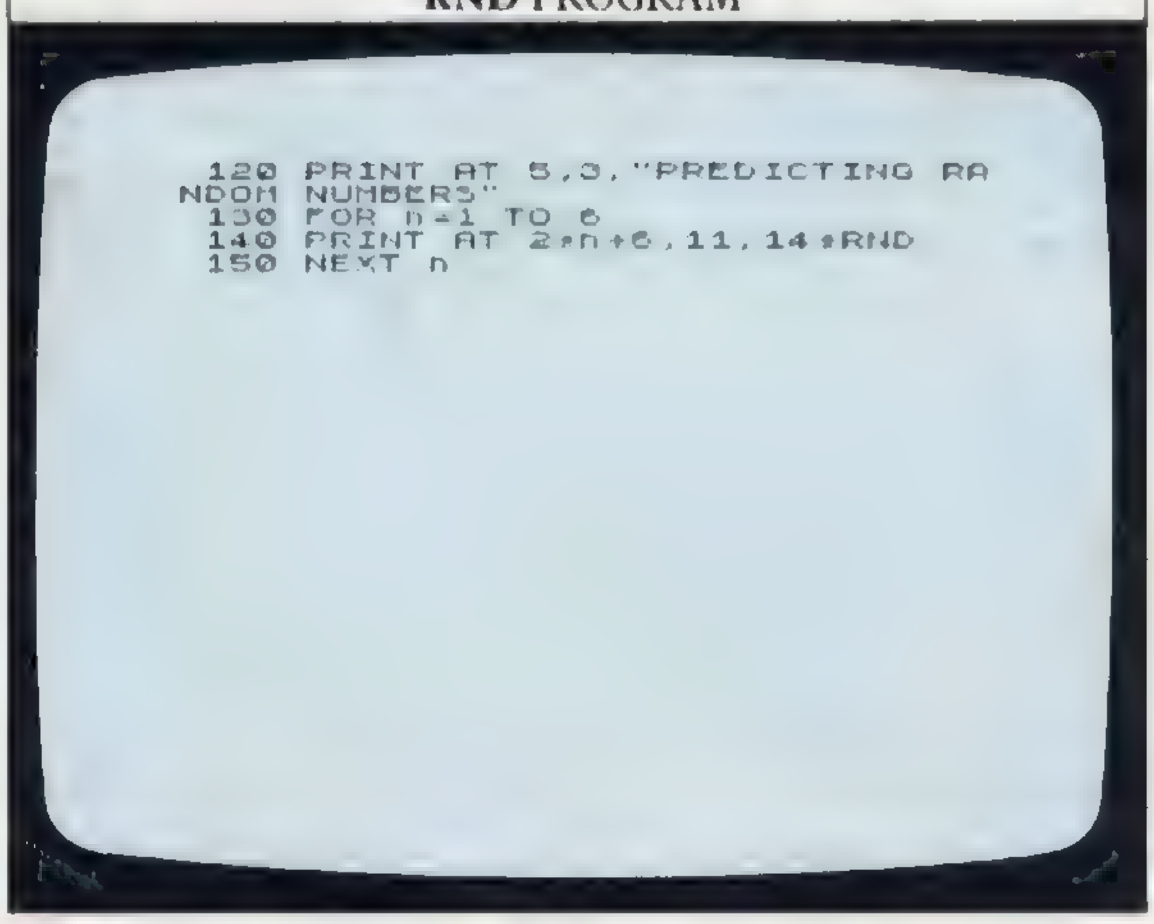
```
100 n=1+INT(6*RND)
```

produces numbers between 1 and 6.

How to start a "random" sequence

Debugging a program that uses RND can be difficult, because no two RUNs of the program produce the same result. It is easier to find the errors if the RND statements are made to produce the same numbers each time the program is RUN. First type in:

RND PROGRAM

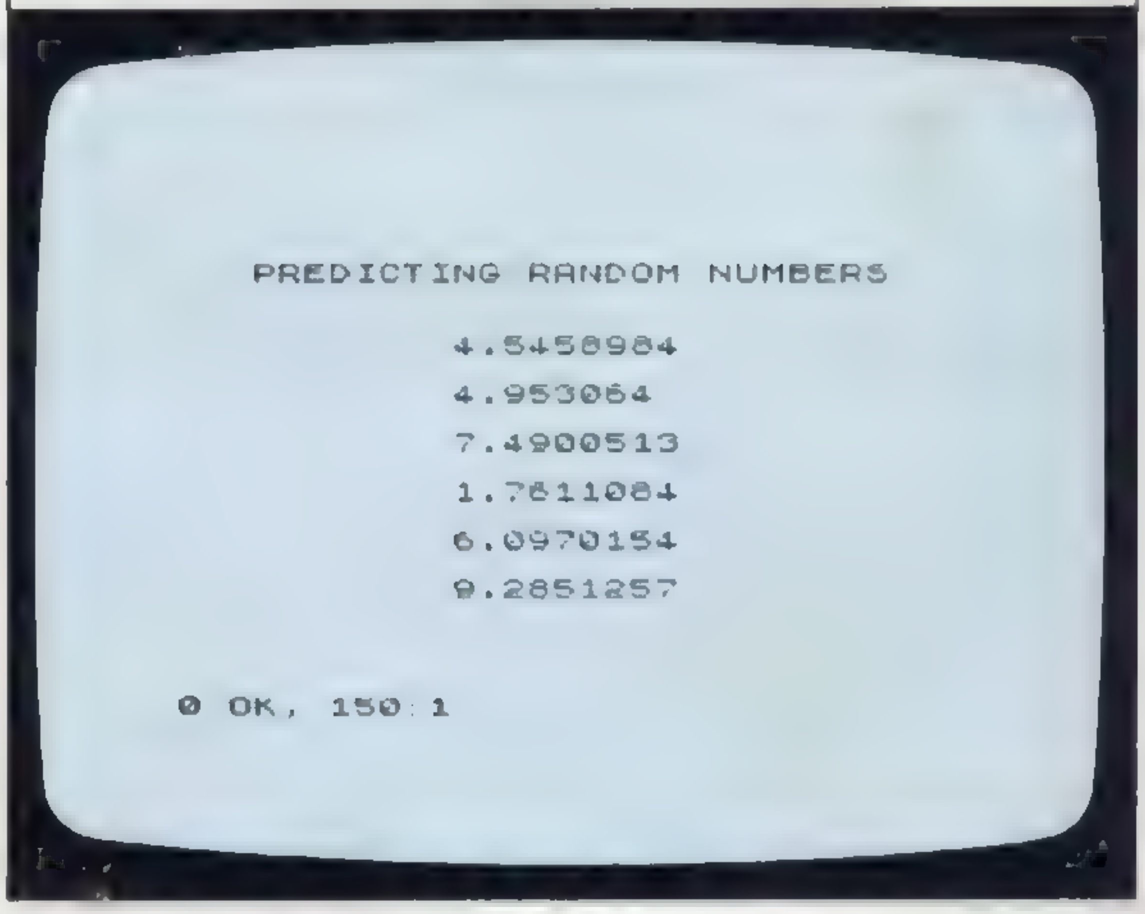


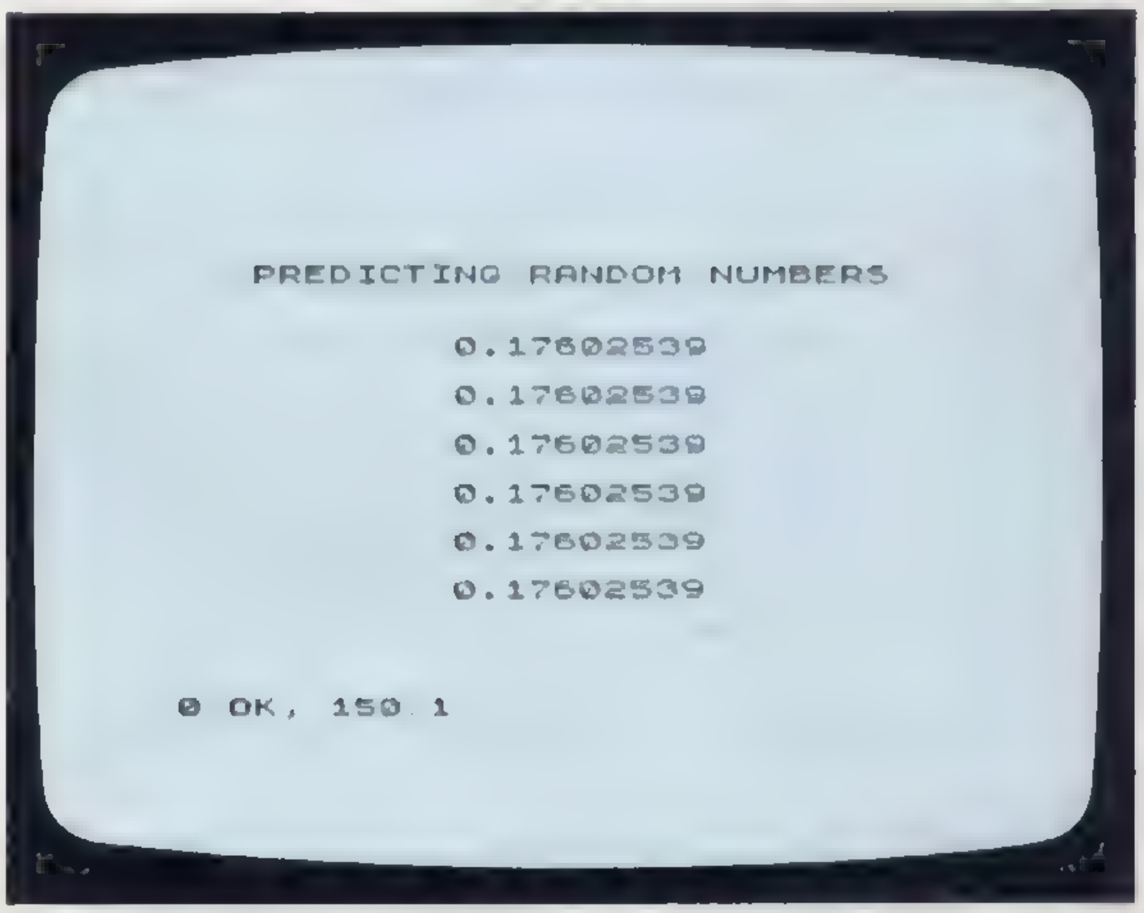
This very simple listing produces a series of six random numbers. Each time you RUN the program you should get a different series of numbers. The first screen below however produces a series which is always identical. It is programmed by adding:

```
125 RANDOMIZE 10
```

By using the keyword RANDOMIZE (which appears as RAND on the keyboard) each RUN produces the same numbers. This command works in a simple way – by controlling where RND starts selecting from its store of 65,536 numbers:

RANDOM/REPEATING DISPLAYS





The second screen shows what happens when you move RANDOMIZE. Using the screen editor, change line 125 to line number 135 and delete 125. Now you should get the same number (0.17602539) every time. RANDOMIZE 10 starts the random sequence off at the tenth stored number on each RUN. You can follow

RANDOMIZE with any number between 1 and 65,536. RANDOMIZE on its own uses the time since the computer was switched on to set the starting point for RND. So, using RANDOMIZE with RND makes RND even more random!

Setting odds in adventure games

Whatever you do with RND, it only produces one number at a time. That's fine for coin-tossing or dice programs where each of the possible results (heads or tails, for instance) is as likely as any other result. But imagine you want to build probability into a program, so that some results are more likely than others. This is how many adventure games are written, making the programs much more interesting than ones that deal with predictable sequences of events.

The following program does this. It demonstrates how, in an adventure game, you can make it likely that the player will encounter some characters or symbols more frequently than others. You can also set the odds so that more often than not, nothing happens:

ADVENTURE ODDS PROGRAM

```
  10 DATA 0,0,1,128,1,64,123,128
,0,0,88,192
  20 DATA 0,0,3,192,3,160,123,12
8,0,0,153,224
  30 DATA 0,0,7,224,1,144,122,12
8,0,64,31,240
  40 DATA 0,0,15,240,1,8,50,128,
12,100,31,240
  50 DATA 7,224,15,240,7,196,2,1
28,18,216,15,240
  60 DATA 3,192,15,240,15,234,2,
128,34,240,15,224
  70 DATA 1,128,7,224,19,144,2,1
28,70,104,7,208
  80 DATA 1,128,1,128,123,128,5,
0,76,68,3,12
  90 FOR n=0 TO 7
 100 READ o,p,q,r,s,t,u,v,w,x,y,
z
 110 POKE USR "a"+n,o
 120 POKE USR "b"+n,p
 130 POKE USR "c"+n,q

scroll?
```

```
 140 POKE USR "d"+n,r
 150 POKE USR "e"+n,s
 160 POKE USR "f"+n,t
 170 POKE USR "g"+n,u
 180 POKE USR "h"+n,v
 190 POKE USR "i"+n,w
 200 POKE USR "j"+n,x
 210 POKE USR "k"+n,y
 220 POKE USR "l"+n,z
 230 NEXT n
 240 BORDER 2: PAPER 6: CLS
 250 PRINT INK 3;AT 5,5;"*******
*****************"
 260 PRINT INK 3;AT 14,5;"******
*******************"
 270 PRINT INK 3;AT 19,14;"*****
"
 280 PRINT INK 0;AT 7,5;"AB";AT
8,5;"CD";AT 7,16;"EF";AT 8,16;"G
H";AT 7,26;"IJ";AT 8,26;"KL"
 290 LET k=0: LET d=0: LET t=0
 300 FOR n=1 TO 100

scroll?
```

```
 300 FOR n=1 TO 100
 310 LET a=INT (20*RND+1)
 320 IF a=1 THEN LET A$="AB": LE
T B$="CD": LET t=t+1
 330 IF a=2 OR a=3 THEN LET A$="
EF": LET B$="GH": LET k=k+1
 340 IF a=4 OR a=5 THEN LET A$="
IJ": LET B$="KL": LET d=d+1
 350 IF a>5 THEN LET A$="**": LE
T B$="**"
 360 PRINT AT 10,16;A$;AT 11,16;
B$
 370 PRINT AT 12,5;t;AT 12,16;k;
AT 12,26;d;AT 17,15;n
 380 BEEP 0.2,15
 390 NEXT n


0 OK, 0:1
```

Lines 10 to 230 reprogram the keys A to L with the 12 characters that make up the three game symbols – a bag of treasure, an armoured knight and a dragon. Each symbol is made up from four characters, the first two above the second two.

Line 310 produces a number between 1 and 20. If it equals 1, then the treasure symbol is selected. If it equals 2 or 3, the knight is selected and if it equals 4 or 5, the dragon is selected. If it equals anything else, nothing (represented by four asterisks) is selected. So, a knight or a dragon are twice as likely to appear as a bag of treasure.

When you RUN the program, 100 random selections are made. The running totals of treasure, knights and dragons are shown building up on the screen. You can change the relative probabilities of the three symbols appearing by changing the numbers in lines 320 to 350. Inserting a PAUSE statement in the loop, or increasing the duration of the BEEP in line 380, slows the program down. By the end of each RUN, you can see how often the program has chosen each of the characters:

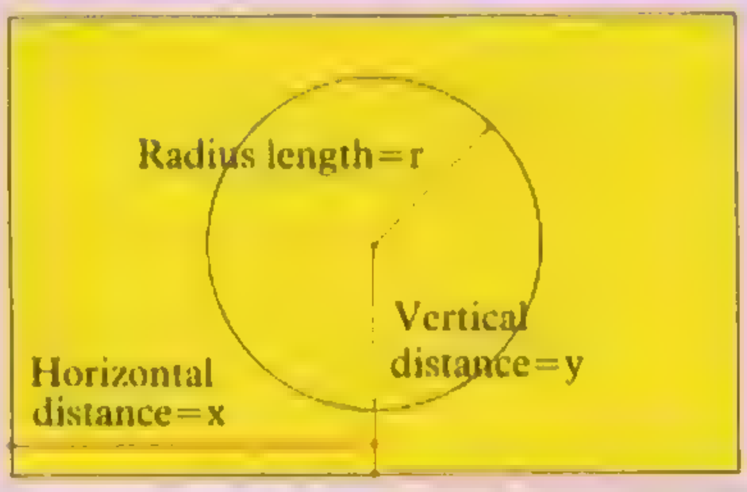
ADVENTURE ODDS DISPLAY

CURVES AND CIRCLES

On many personal computers, drawing curves and circles is quite an involved process. The Spectrum makes drawing circles very easy – its CIRCLE command does all the mathematics that you would otherwise have to do yourself. To produce a circle, all you have to do is specify the co-ordinates of the point at the centre of the circle – x and y, and the length of its radius, r. This lets you produce any size of circle in any position, within the limits of the graphics grid.

POSITIONING A CIRCLE

Any circle can be produced with the command CIRCLE x,y,r



The following direct command will produce a circle on the screen:

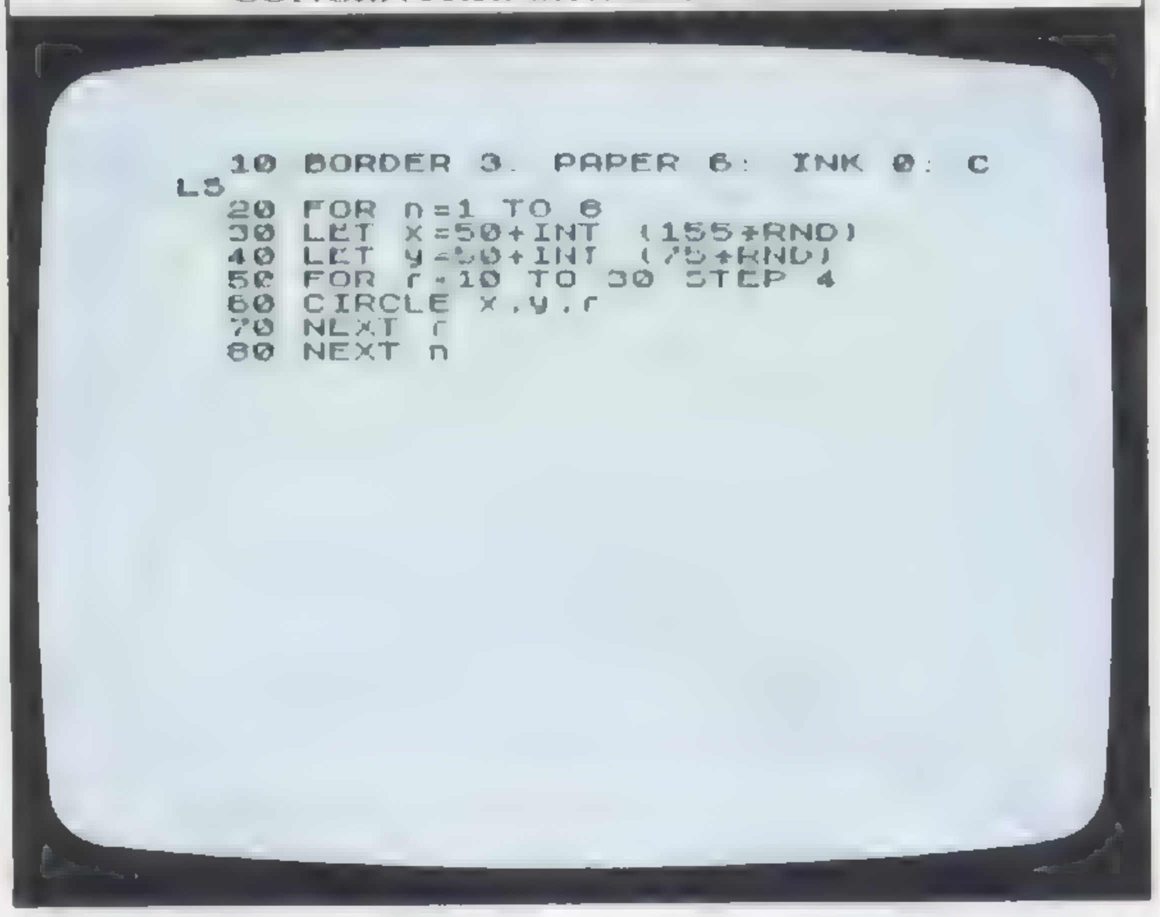
```
CIRCLE 128,88,50
```

The centre of this circle is at the centre of the screen (128,88) and it has a radius of 50 units. So, the horizontal diameter goes from 78 to 178 units across the screen and the vertical diameter from 38 to 138 up it.

Producing patterns with CIRCLE

You can use the CIRCLE command to make up patterns by selecting the co-ordinates of a circle's centre at random. Here is a program that builds up a display by this method:

CONCENTRIC CIRCLES PROGRAM



```
10 BORDER 3: PAPER 6: INK 0: C
LS
20 FOR n=1 TO 6
30 LET x=50+INT (155*RND)
40 LET y=50+INT (75*RND)
50 FOR r=10 TO 30 STEP 4
60 CIRCLE x,y,r
70 NEXT r
80 NEXT n
```

CONCENTRIC CIRCLES DISPLAY

Lines 30 and 40 produce a pair of x and y co-ordinates at random so that neither is within 50 units of any of the screen edges. It is set like this because the program will then draw concentric circles up to a radius of 50. The loop at lines 50 to 70 repeatedly draws circles with the same centre but with gradually increasing radii (r). Line 80 starts the whole process off again but with a new pair of random co-ordinates.

You can change the maximum radius of the circle and the STEP size between radii by changing line 50:

```
50 FOR r=10 to 20 STEP 3
```

or even STEP 2, which produces smaller patterns.

Drawing arcs and waves

An arc (part of the edge of a circle) can be produced on the Spectrum's screen, but you cannot use the CIRCLE command to do this. But adding another number to the DRAW command – making it DRAW x,y,a – DRAWs an arc of a circle, starting from the last point PLOTted or DRAWn to, and finishing at a point specified by x,y. The "a" value is more complicated. This is the angle that the arc turns through (if you imagine it as being part of a circle). You can see the command at work if you key in these two lines (you will find the keyword PI above the M key):

```
PLOT 10,88
DRAW 230,0,PI/4
```

This produces a display like a piece of rope suspended at both ends and sagging in the middle. The first statement PLOTs a point near the left edge and halfway up the screen. The DRAW statement then DRAWs an arc from that point in an anticlockwise direction to (10+230),(88+0) or 240,88.

Whether the arc forms a shallow sag or a semicircle is determined by the size of "a". This angle is measured not in degrees but in radians. Radians are simply an alternative way of measuring angles based on the geometry of a circle. A complete circle is produced by turning around through 360 degrees. That is exactly equivalent to 2*PI radians. PI is an important mathematical constant, which has a value of 3.14159625 … (you can see this by keying in PRINT PI). It is the ratio of the length of the circumference of a circle to its diameter. There are 2*PI radians in a circle. PI/2 radians are therefore equivalent to a quarter of a circle (90 degrees), PI/4 radians are equivalent to one eighth of a circle (45 degrees) and so on.

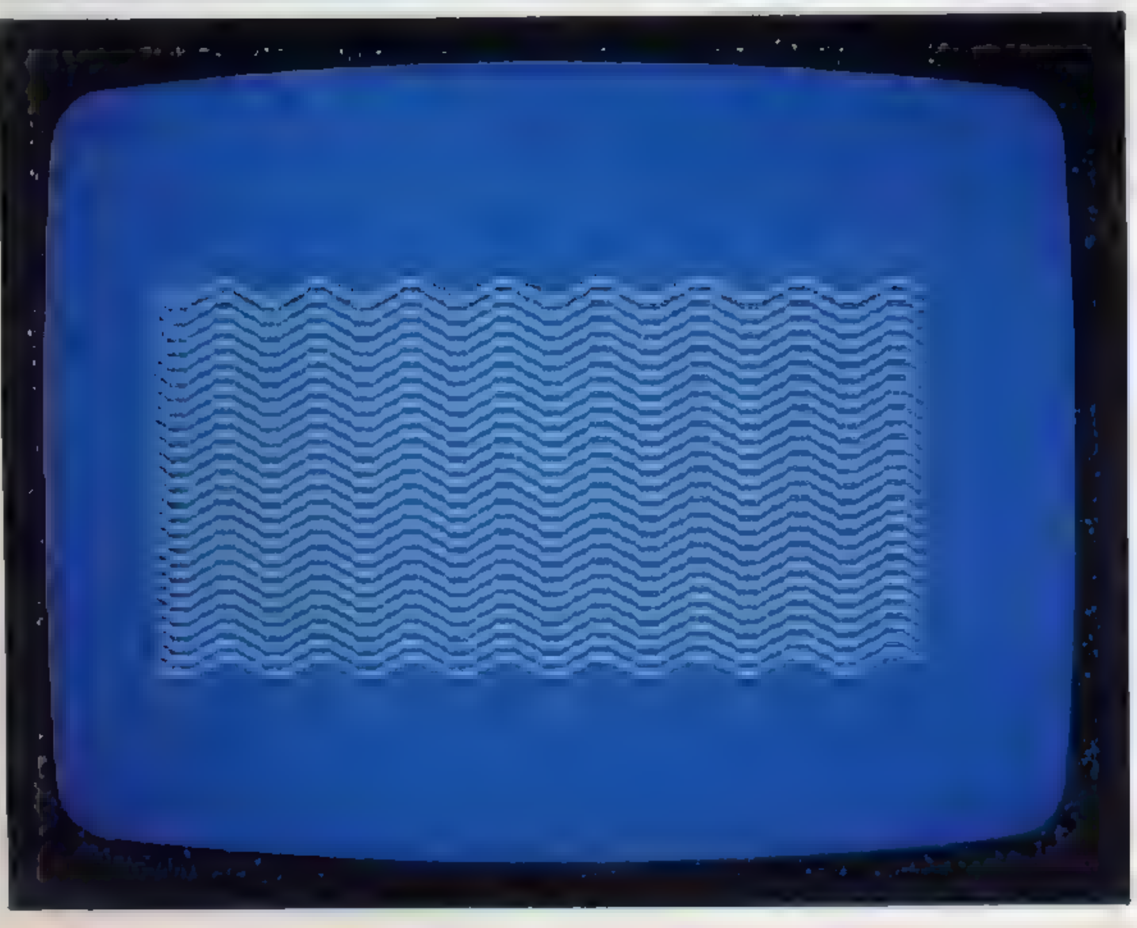
Now, without erasing the first arc, try adding:

```
PLOT 10,88
DRAW 230,0,-PI/4
```

The second arc is DRAWn between the same two points to form an eye-shaped figure. You can build up shapes using arcs to produce wave-like patterns:

WAVE PATTERN PROGRAM

```
  10 BORDER 1: PAPER 1: INK 7: C
LS
  20 LET y=120
  30 FOR i=1 TO 26
  40 FOR x=0 TO 225 STEP 30
  50 PLOT x,y
  60 DRAW 15,0,PI/2: DRAW 15,0,-
PI/2
  70 NEXT x
  80 LET y=y-5
  90 NEXT i

0 OK, 0:1
```



Line 20 sets the first y co-ordinate value of the waves. Line 40 STEPs the x co-ordinate across the screen. It's important that the STEP size is the same as the combined x lengths of the two parts of the wave formed by each loop. Line 50 PLOTs a point at the x,y co-ordinates set at the beginning of each loop. Line 60 DRAWs a small arc of a circle and then it goes on to DRAW a second arc of the same size. The first arc sags downwards, the second upwards and so on, building up a row of waves. When that is complete, line 80 moves the y co-ordinate downwards. The program continues until the waves reach the bottom of the screen.

Programming creeping curves

On the previous page CIRCLE was used to produce concentric circles at random. Here is a program that DRAWs semicircles, also at random, each of which goes up, down, left or right, to creep over the screen:

CREEPING CURVES PROGRAM

```
  10 BORDER 1: PAPER 6: INK 0: C
LS
  20 PLOT 128,88
  30 LET x=0: LET y=0
  40 LET q=1+INT (4*RND)
  50 IF q=1 THEN LET x=10
  60 IF q=2 THEN LET x=-10
  70 IF q=3 THEN LET y=10
  80 IF q=4 THEN LET y=-10
  90 DRAW x,y,PI
 100 GO TO 30

0 OK, 0:1
```

Try changing the arc size in lines 50 to 80 for different effects. You'll find that if you reduce the arc size to 4, it looks more like a letter v.

NATURAL GRAPHICS

Most of the shapes you have DRAWn so far have involved working out beforehand where the key points of the shape (the corners of a square, for example) should lie, and then DRAWing lines between them. Fortunately, much more complicated geometrical patterns can be DRAWn by using functions that do all the work for you. You don't have to PLOT a single point. Using the functions' SIN (short for the mathematical term "sine") and COS (short for "cosine"), you can produce some spectacular graphics with quite short programs.

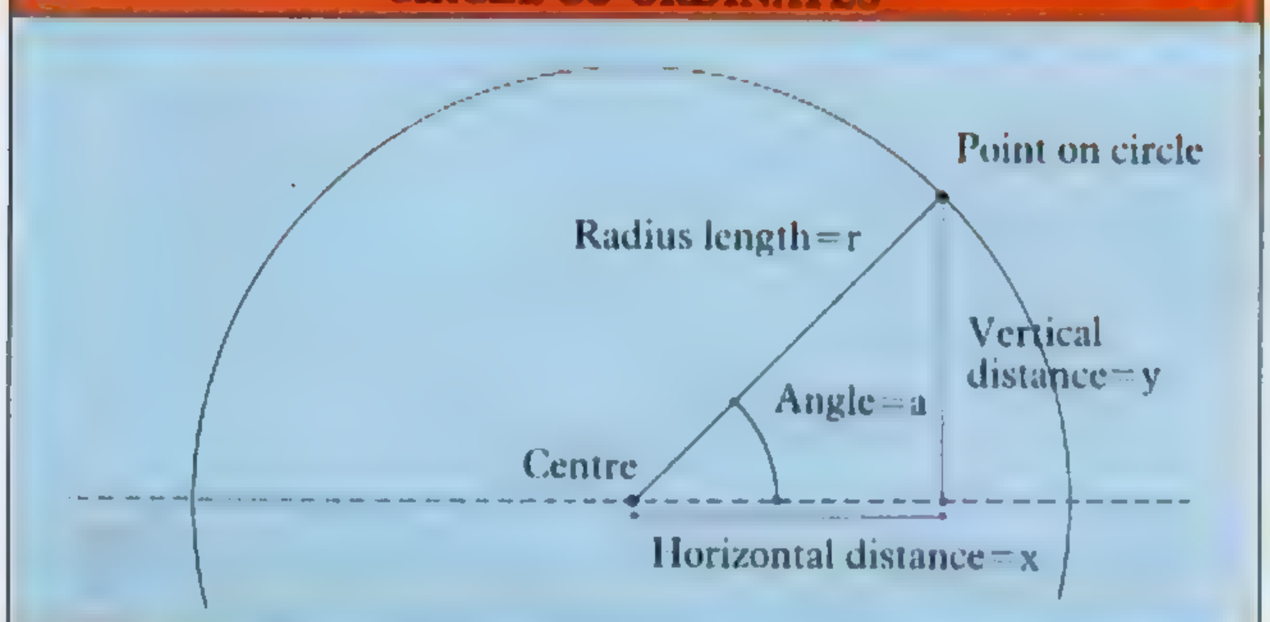
All of the programs on these two pages use a combination of SIN and COS, so it's worth trying to understand what these commands actually mean.

Another way to PLOT circles

On page 16, you saw how the command CIRCLE can be used to produce a circle. You can actually PLOT a circle by another method, using SIN and COS. Although the program is longer, it then enables you to branch out into some much more elaborate graphics.

If you sketch out part of a circle, you can relate each point on the circle to an angle at the circle's centre.

CIRCLE CO-ORDINATES



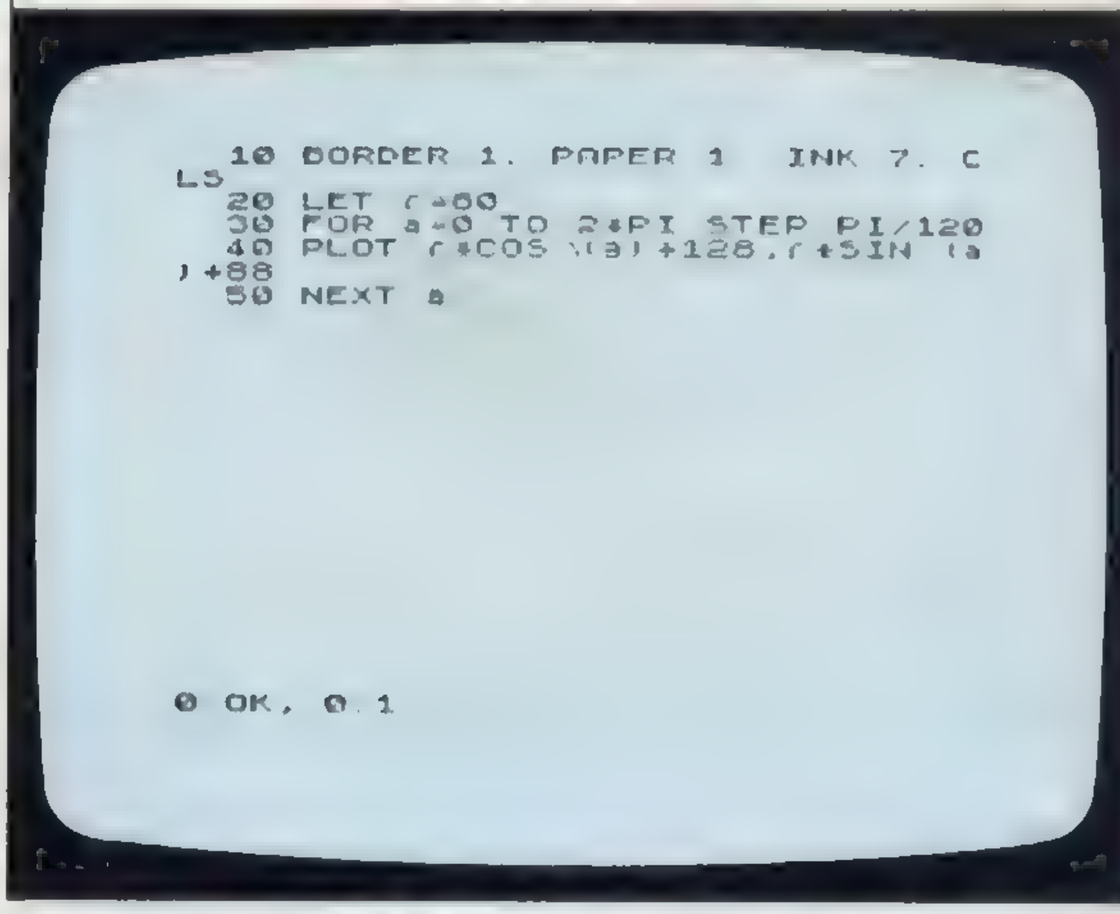
The distances x and y can be expressed in another way, as multiples of the angle and SIN or COS. Every angle has its own value of SIN and COS, and the co-ordinates of any point on the circle can be written as:

r*COS(a),r*SIN(a)

Once you know this, you can get the computer to PLOT a circle. By working through all the possible values of the angle "a" with a loop, a program can use the functions SIN and COS to PLOT the co-ordinates of every point on the circle.

You can, of course, do this much more easily by using CIRCLE. However, if you key in the next program, you can develop it to produce graphics that CIRCLE itself cannot produce. Line 20 sets the radius of the circle to 80. The angle has to vary from zero to a full circle (360 degrees):

CIRCLE PROGRAM



Because the BASIC functions SIN and COS operate on angles measured in radians, like the command DRAW x,y,a on page 16, the range of the angles again uses the keyword PI. The STEP of PI/120 (line 30) is chosen so that the dots are relatively close together but the program doesn't take too long to RUN.

Patterns with SIN and COS

The circle program, above, produces a circle because the x and y co-ordinates vary exactly out of step with each other. When x is zero, y is at its maximum value and vice versa. What would happen if you varied x and y at different rates? Try altering the angle after the SIN command. Here the angle has been doubled in the first display, and multiplied by five in the second:

"LISSAJOUS" FIGURE – **SIN** (2*a)

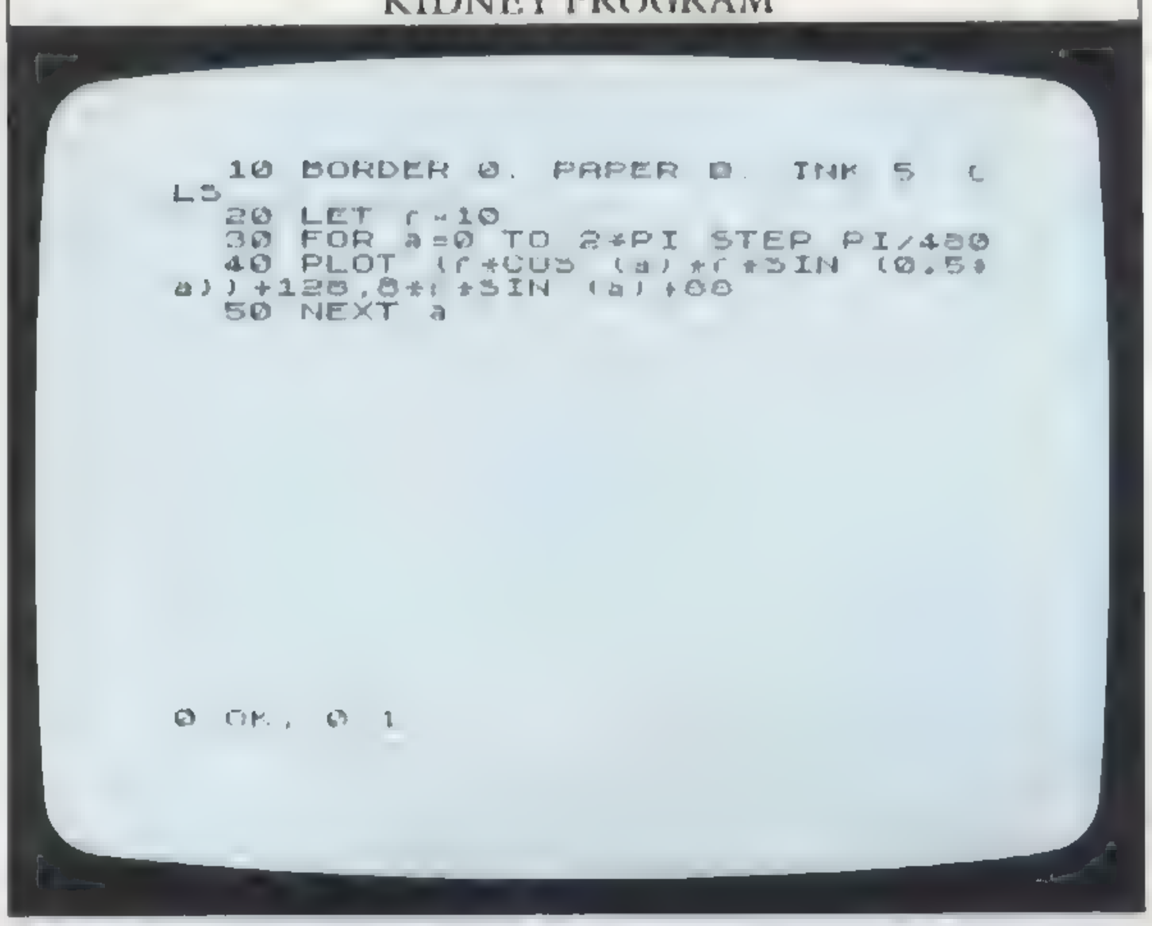
The number of loops in each display depends on how many times the angle after SIN has been multiplied:

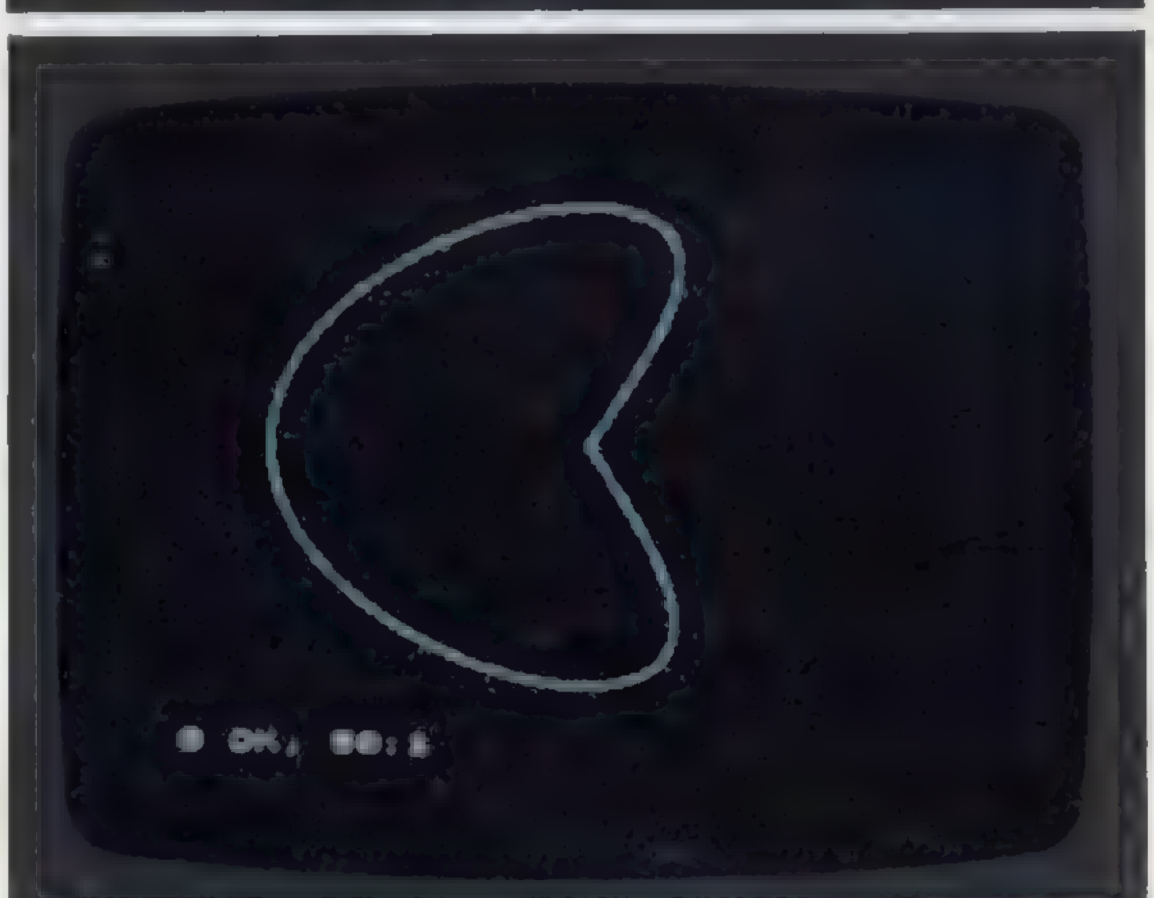
"LISSAJOUS" FIGURE – **SIN** (5*a)

Complex curves

Now you can make a different sort of change:

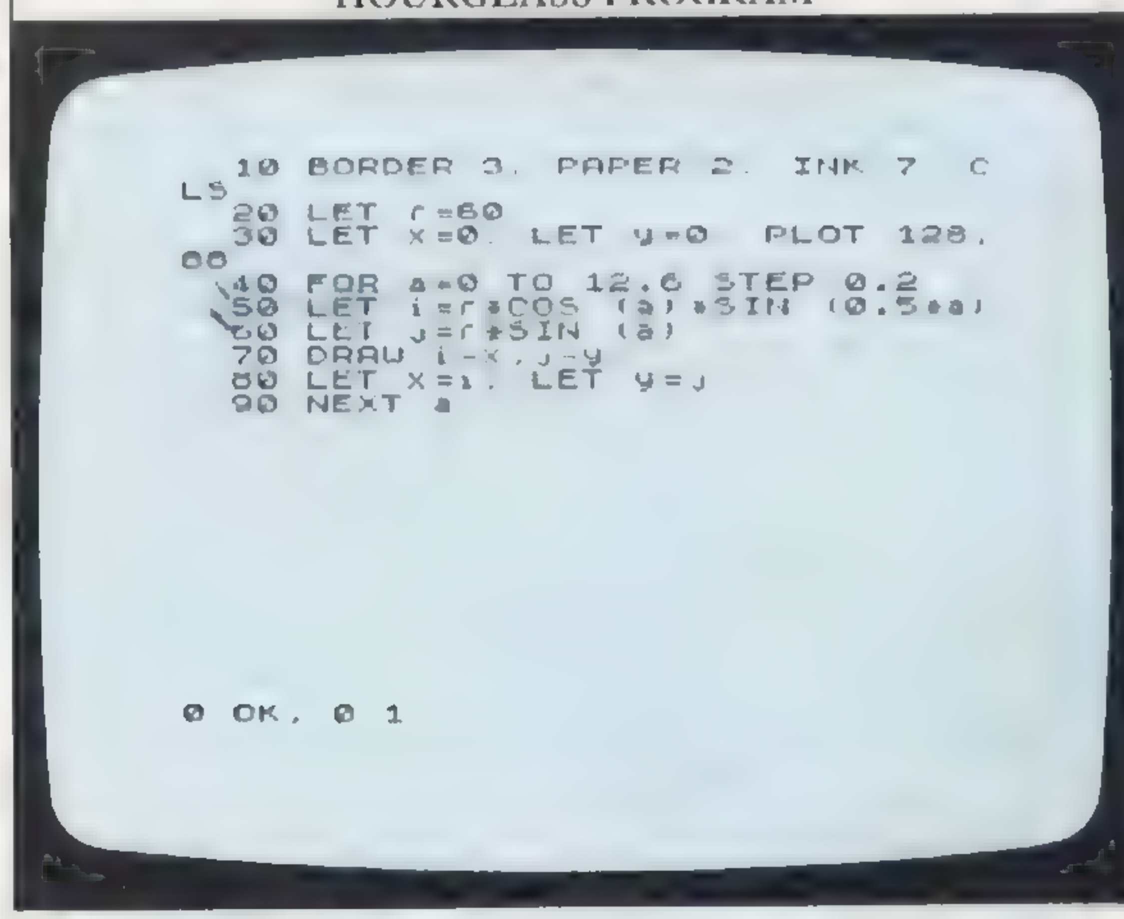
KIDNEY PROGRAM





In this program, the colours and STEP size have been changed as well as the co-ordinate values. The program will continue DRAWing if you increase the range of angle values. The next program roughly doubles the range. It also accelerates the RUNning speed by DRAWing short lines instead of PLOTting individual points as in the CIRCLE program opposite:

HOURGLASS PROGRAM



You can experiment with this program to produce a whole variety of more complicated shapes. The next design is produced by changing the colours and lines 40 and 50 to:

```
40 FOR a=0 TO 1000 STEP 0.1
50 LET i=r*COS(a)*SIN(0.98*a)
```

The shape starts off by being quite open but eventually the program fills in the circle (this screen shows it after half an hour of RUNning):

BALL DISPLAY

You may find with this type of program that the graphics resolution cannot cope with too much detail as every curve has to be shown as a series of dots.

THE COMPUTER CLOCK

Time delays are among the most frequently used commands in computer programs. One of the problems of working with a micro is that it often RUNs through programs too quickly. Time delays slow programs down to a speed that is useful. A delay may keep some text on the screen for a moment, long enough to read it, before it disappears and the program continues, or it may slow down an otherwise too rapid sequence of images – in an animation program for example.

The quickest and easiest time delay to write into a program is the PAUSE statement. It takes the form of the following program line:

```
100 PAUSE 50
```

where the number after PAUSE represents the time delay in fiftieths of a second. It's a simple matter to RUN a program containing the above PAUSE statement and see for yourself if the delay is long enough. If not, then increase and test the number following PAUSE again. It's approximate, but convenient. However, if you want to time an event in a program you can use a clock that has been built into the Spectrum itself.

Using the internal clock

Some computers have a timing facility, a clock that runs regardless of whatever the user's programs are doing. The Spectrum doesn't have any command or function that enables you to use a timing facility with a single program statement.

However, in common with every other personal computer, the Spectrum has an internal clock which synchronizes its activities. Linked to this is a counting device. Three "addresses", or slots, in the Spectrum's memory count the number of television pictures (or frames) that have been produced since the computer was switched on. The first address (numbered 23672) counts frames up to a maximum of 255. Then it is reset to 0 and the contents of the next address (23673) are increased by 1. When that reaches 255, it is reset to 0 and the contents of the third address (23674) are increased by 1. Because, in Europe, television pictures are produced at the rate of 50 per second (the rate is 60 per second in the United States), the first address (23672) in effect counts in fiftieths of a second, the next address in 5.12-second intervals and the third address in 1310.72-second intervals.

If this internal clock/counter is only accurate to a fiftieth of a second, you might wonder what the advantage is of using it instead of the much more convenient PAUSE statement, which has roughly the same order of accuracy. The reason is that PAUSE is a BASIC statement which makes up part of a program. The program is unable to do anything else during the PAUSE. It can never be more than a temporary interruption to the program.

The frame counter works in a different way. It keeps on counting whatever the program is doing – with two exceptions. If you make the computer BEEP or you use any piece of hardware connected to the computer – a cassette recorder or printer, for example – the frame counter stops while you are doing so, and then resumes its count. So, if you want to use the frame counter as a clock, it would lose time during these operations.

Timing with the frame counter

To see the frame counter working, try keying in the following two lines:

```
10 PRINT PEEK 23672
20 GOTO 10
```

The screen should fill up with increasingly large numbers (up to 255), until the "scroll?" prompt appears, as shown on the display:

FRAME COUNTER DISPLAY

PEEK looks into the address specified to see what number is stored there. It's the counterpart of POKE, which puts a number into an address. PEEK 23672 means "the contents of" address number 23672. Note that the numbers PRINTed aren't consecutive. There is a difference of three or four between each pair. That's because the PRINT and GOTO statements themselves take some time to be carried out, so the program isn't able to "catch" every single fiftieth of a second "tick" of the frame counter. That can cause problems when you want to use the frame counter to count in small time intervals, as you can see from the following program. When you RUN it, a BEEP should sound every second. The variable t is set equal to the contents of 23672:

FRAME COUNTER "CLOCK"

```
  10 LET t=PEEK 23672
  20 IF PEEK 23672-t=50 THEN BEE
P 0.05,20: GO TO 10
  30 GO TO 20

0 OK, 0:1
```

When the difference between the PEEK 23672 and the fixed t reaches 50, then one second has passed and the BEEP should be sounded.

Unfortunately, as you probably discovered, when you RUN this program, it hardly produces more than a couple of BEEPs. The trouble is that PEEK 23672 – t is unlikely to be equal to 50 at the instant when line 20 is being carried out. Also, the BEEP statement stops the frame counter for a twentieth of a second every time it sounds. This sort of program can be used as a timer, but the time interval (the time between ticks) must be large compared to the time taken to carry out the program statements.

The next program times a 5-second interval, although it is not completely accurate. The second address counts every time the first address is full – that is, every 256 fiftieths of a second, or 5.12 seconds. That is adequate for a timer that only has to be accurate to the nearest minute or half minute. Line 10 resets the address to 0 every time the program is RUN:

5-SECOND TIMER

```
  10 POKE 23673,0
  20 LET t=PEEK 23673
  30 IF PEEK 23673>t THEN PRINT
"tick"
  40 GO TO 30

0 OK, 0:1
```

Using time in programs

Because PEEKing the frame computer requires a number of calculations, you can substitute a function FN t() to represent the number of seconds that have elapsed since the computer was switched on. A variable T is set equal to this, and, later in the program, when FN t() exceeds T, the seconds display is increased by 1. T is then reset to FN t() for the next count. A program can also incorporate tests to increase the minutes and hours displays when necessary.

This relatively accurate timing is of more value in programs. Here is an example of one way to use it, in a program that times the speed of your reactions:

REACTION TIMER

```
  10 DEF FN t()=65536*PEEK 23674
+256*PEEK 23673+PEEK 23672
  20 PRINT AT 5,6;"Time your rea
ctions"
  30 PRINT AT 10,10;"Press a key
"
  40 PRINT AT 12,4;"when you hea
r the tone"
  50 PAUSE 100+4*(INT 100*RND)
  60 LET T=FN t(): BEEP 0.1,25
  70 PRINT AT 16,6;"Your time st
arts now"
  80 IF INKEY$="" THEN GO TO 80
  90 CLS
 100 PRINT AT 8,12;"You took";AT
 10,13;(FN t()-T)/50;AT 12,12;"s
econds"
 110 PAUSE 150
 120 CLS : GO TO 20

0 OK, 0:1
```

```
      Time your reactions


          Press a key
    when you hear the tone


      Your time starts now
```

The time function FN t() is defined in line 10. It PEEKs all three frame counters. The final division by 50 is omitted, so that the function counts in fiftieths of a second, instead of in seconds. Line 50 causes a random delay of at least 2 seconds before the timing period starts. Line 80 waits until you press a key before clearing the screen (line 90) and then calculating and PRINTing your reaction time.

USING ARRAYS

An array is a way of storing facts and/or figures in the computer's memory in the form of a table, so that you can locate any one item in the table without having to go through all the others first. Each item in an array is specified by one or more numbers. In the following array, each item is given a pair of co-ordinates which identify it and nothing else:

	1	2	3	4	5	6
1	FRED	KATE	JOHN	JANE	ALAN	JUDY
2	100	250	840	125	223	691

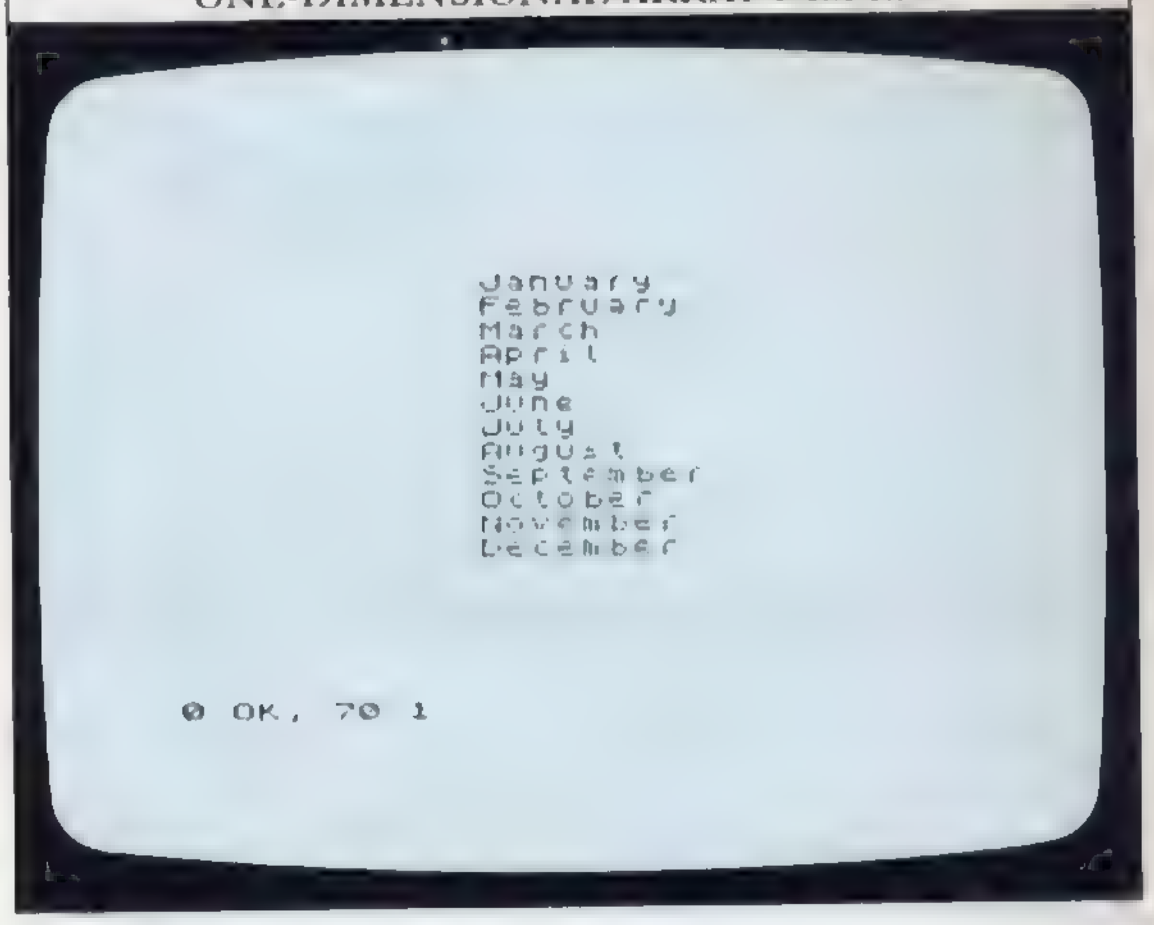
This is a 6×2 array, so-called because it has 6 columns and 2 rows. Item (2,2) is 250, item (1,3) is JOHN, and so on. Because two numbers are needed to identify each item, this array is known as a two-dimensional array. If it was composed of only one row of names or numbers, it would only need one number to identify each item and so it would be called a one-dimensional array. The BASIC keyword DIM is used to tell the computer how big an array is to be.

A one-dimensional array can be used to store a list of frequently used numbers or strings:

ONE-DIMENSIONAL ARRAY

Here line 20 tells the computer that the array m$ is 12×9 entries (9 is the maximum number of letters in the name of any one of the 12 months). The program PRINTs out the list of the months of the year given in line 10. Although there are easier ways of doing this, later on in a program, you might want to match up a month with other information or the result of calculations. Using this listing, you can pick out any month by using m$(n) where n is the month number. When the program is RUN, the display it should produce looks like this – a month chart ready for more information:

ONE-DIMENSIONAL ARRAY DISPLAY

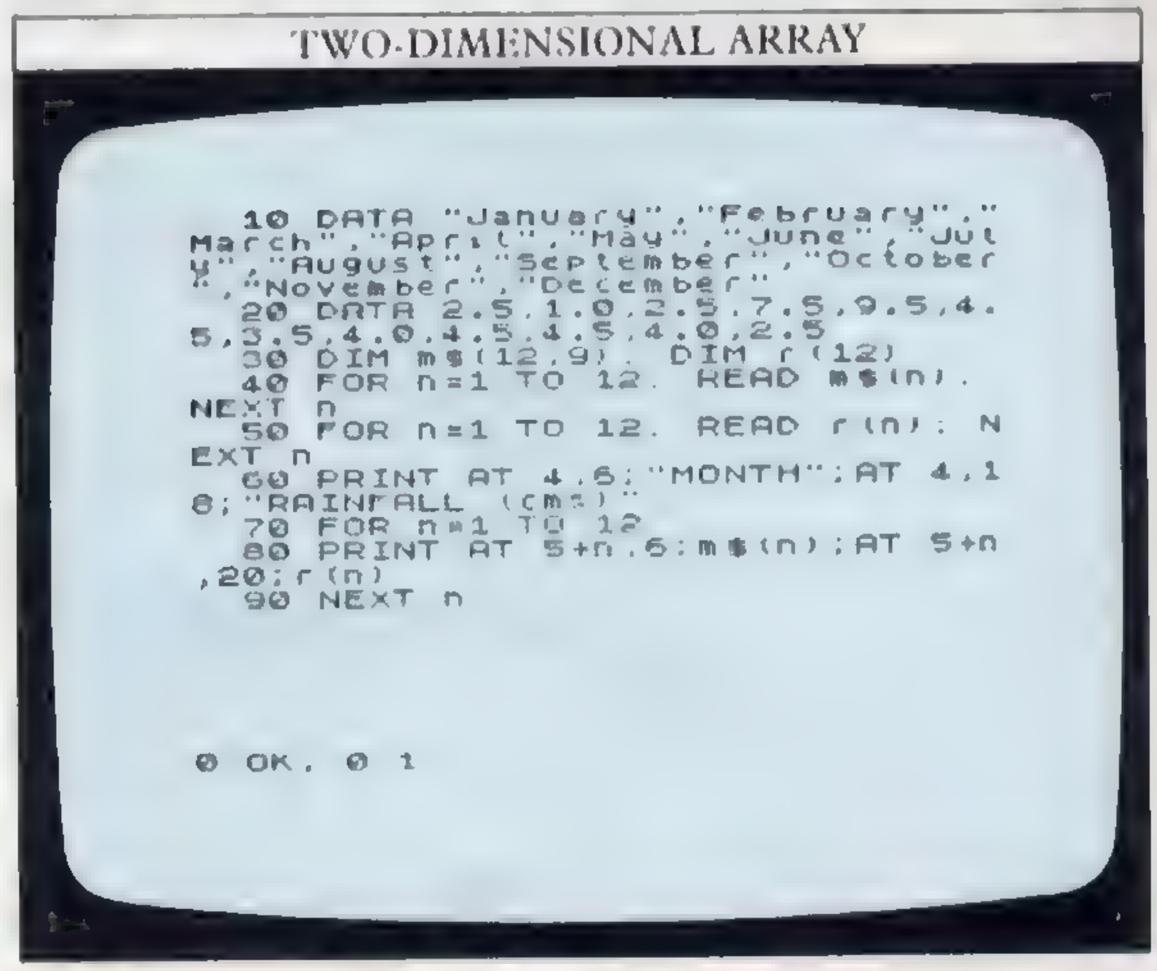


How to add a dimension

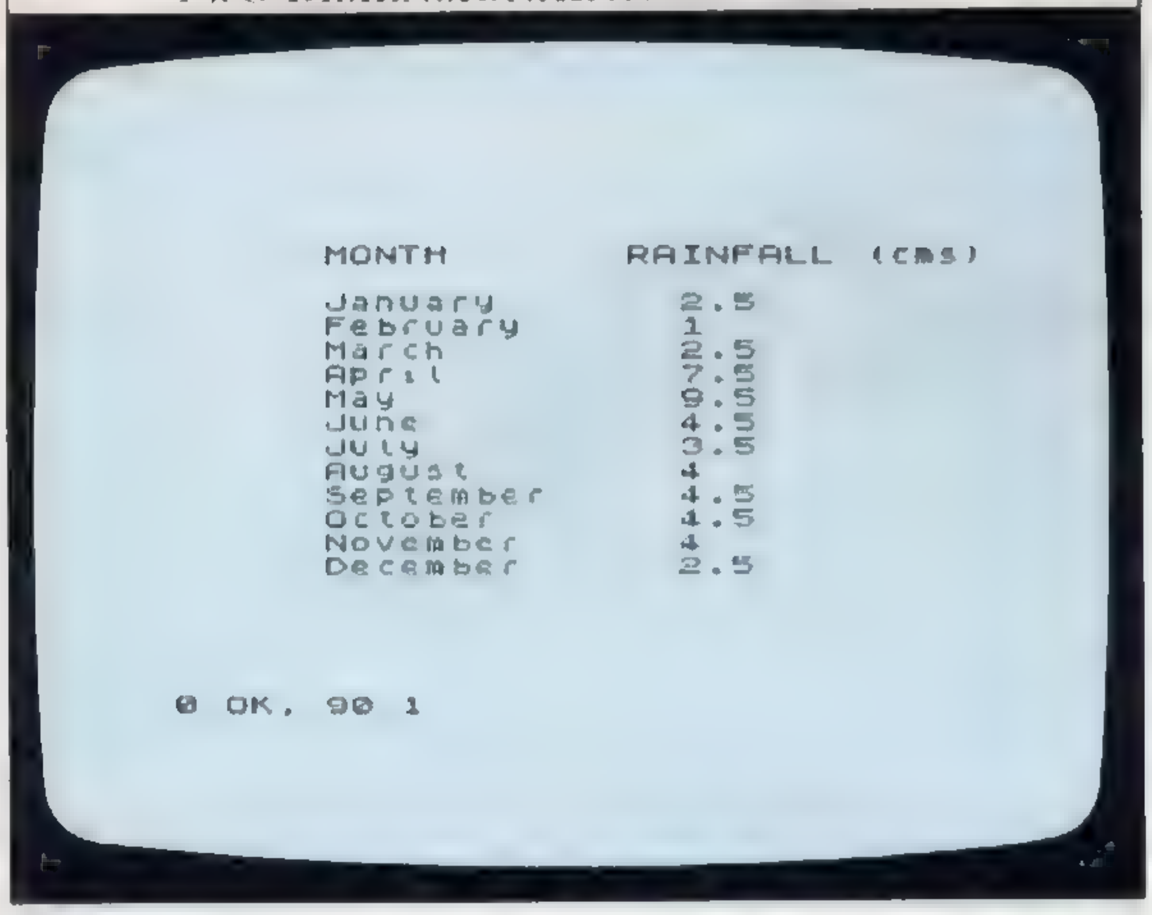
Now you build upon the calendar array program to make it do something useful. Add a second array, a numeric array, so that you can list some values against each month. The following program uses the array to represent monthly rainfall totals.

The table now has two headings. You don't have to PRINT all the members of the string array – m$(n) – before moving on to select the numeric array – r(n). Line 80 takes one item from each array. As these are to be PRINTed on consecutive rows of the screen, they can easily be identified by relating them to the row number. For each value of n, m$(n). and r(n) are PRINTed at different column numbers of row (5+n):

TWO-DIMENSIONAL ARRAY



TWO-DIMENSIONAL ARRAY DISPLAY

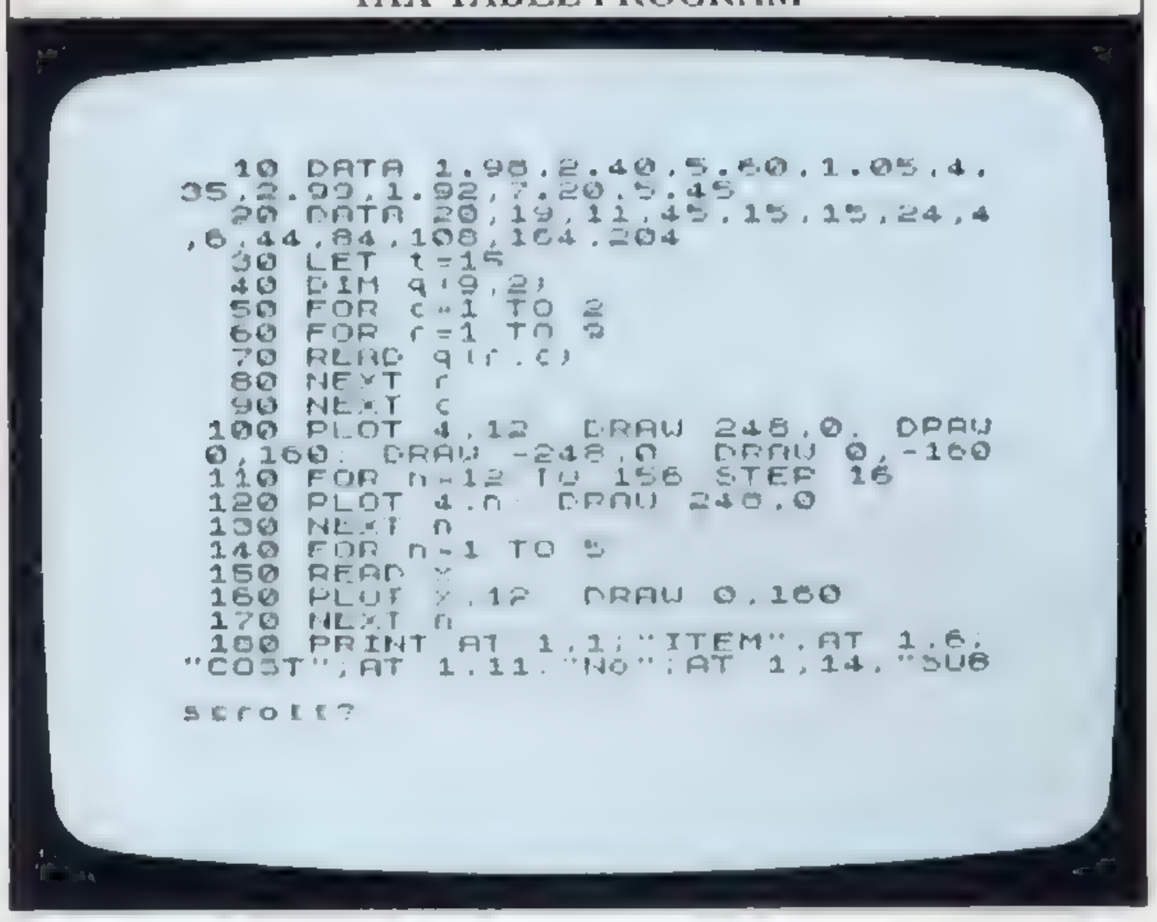


```
MONTH         RAINFALL (cms)

January       2.5
February      1
March         2.5
April         7.5
May           9.5
June          4.5
July          3.5
August        4
September     4.5
October       4.5
November      4
December      2.5



0 OK, 90 1
```

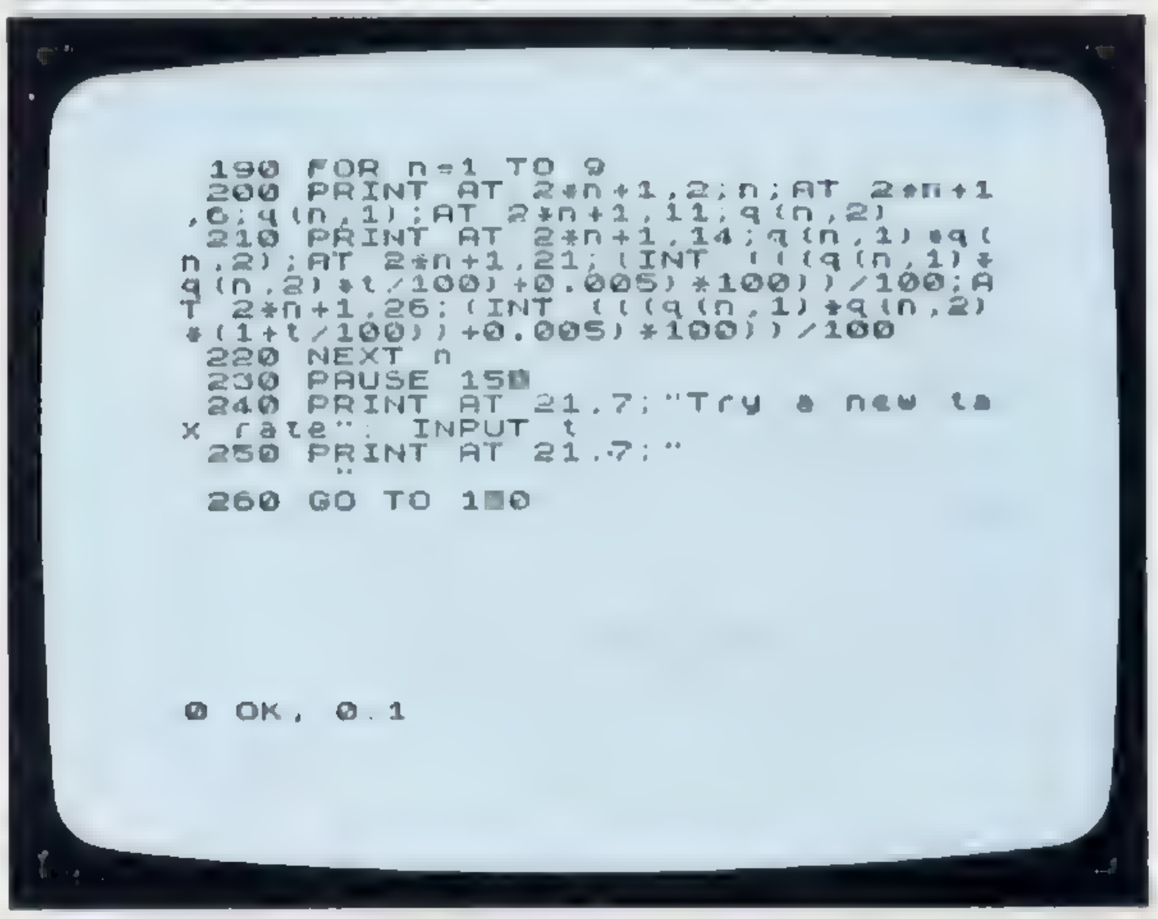
Writing tables with arrays

You can now make a table that is more ambitious:

TAX TABLE PROGRAM



```
  10 DATA 1.98,2.40,5.60,1.05,4.
35,2.99,1.92,7.20,5.45
  20 DATA 20,19,11,45,15,15,24,4
,6,44,84,108,164,204
  30 LET t=15
  40 DIM q(9,2)
  50 FOR c=1 TO 2
  60 FOR r=1 TO 9
  70 READ q(r,c)
  80 NEXT r
  90 NEXT c
 100 PLOT 4,12: DRAW 248,0: DRAW
0,160: DRAW -248,0: DRAW 0,-160
 110 FOR n=12 TO 156 STEP 16
 120 PLOT 4,n: DRAW 248,0
 130 NEXT n
 140 FOR n=1 TO 5
 150 READ x
 160 PLOT x,12: DRAW 0,160
 170 NEXT n
 180 PRINT AT 1,1;"ITEM";AT 1,6;
"COST";AT 1,11;"No";AT 1,14;"SUB

scroll?
```



```
 190 FOR n=1 TO 9
 200 PRINT AT 2*n+1,2;n;AT 2*n+1
,6;q(n,1);AT 2*n+1,11;q(n,2)
 210 PRINT AT 2*n+1,14;q(n,1)*q(
n,2);AT 2*n+1,21;(INT (((q(n,1)*
q(n,2)*t/100)+0.005)*100))/100;A
T 2*n+1,26;(INT (((q(n,1)*q(n,2)
*(1+t/100))+0.005)*100))/100
 220 NEXT n
 230 PAUSE 150
 240 PRINT AT 21,7;"Try a new ta
x rate": INPUT t
 250 PRINT AT 21,7;"
 "
 260 GO TO 180



0 OK, 0.1
```

In this financial planning program, the columns are interrelated and you have the option of changing some of the information displayed, should you need to do so.

Line 10 contains the DATA for the first part of the array, a series of prices, and line 20 the DATA for a second part – a series of quantities. Line 20 also contains some co-ordinates which will be used later in the program. Lines 40 to 90 dimension the 9×2 array and READ in its DATA. Lines 100 to 170 simply DRAW the grid of lines that frames the DATA. The co-ordinates of the bottom end of the vertical lines in the grid are stored in line 20. Line 180 PRINTs the column headings.

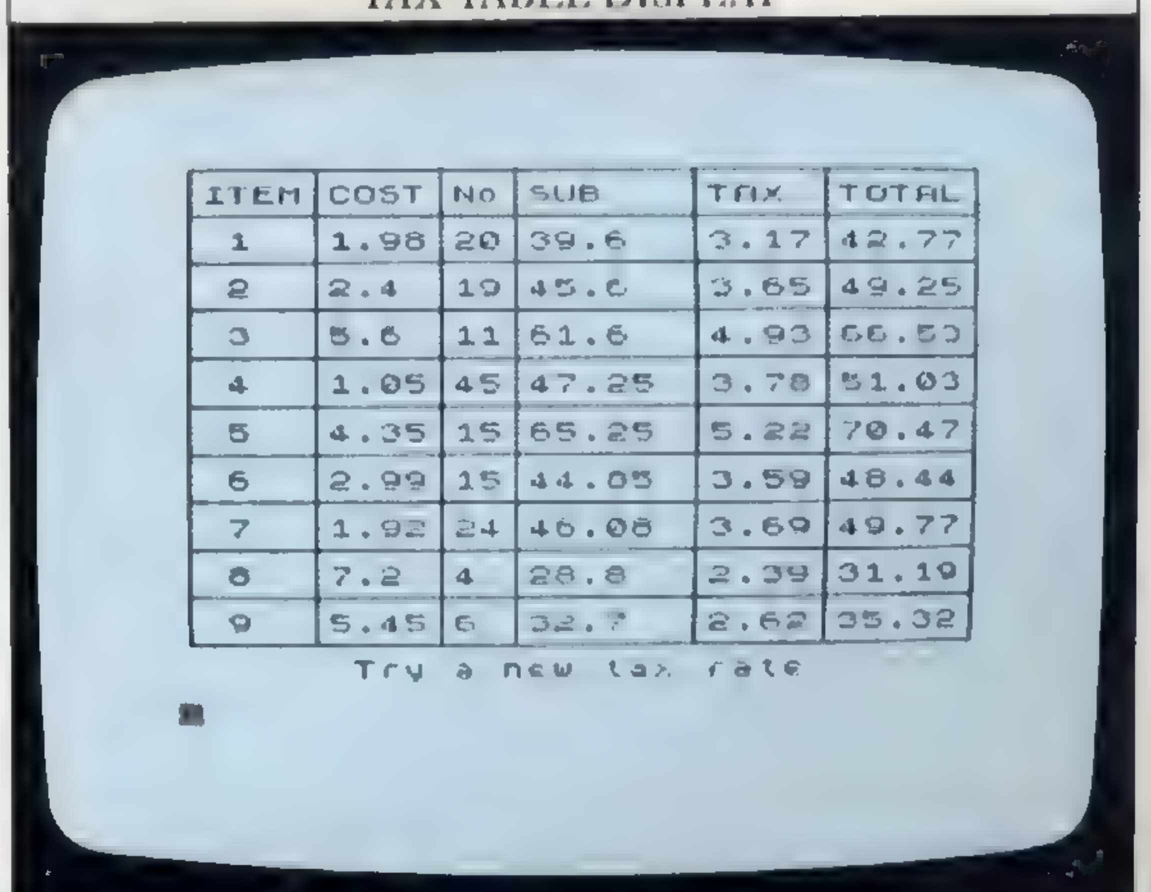
The DATA is PRINTed in the grid by lines 190 to 220. It is to be PRINTed every other line from rows 3 to 19. The number of the item, n, (from 1 to 9) is related to this by:

row=2*n+1

and this appears throughout lines 200 and 210 in the PRINT AT statements. The last two items in line 210 look particularly complex. If the subtotal was 8.25, the tax would be calculated as 0.15*8.25=1.2375 – too many decimal places. To solve that, the tax is multiplied by 100, the INTeger value of it is taken (removing all the decimal places) and it is divided by 100 again. The 0.005 is added to ensure that the final figure is rounded down to the nearest unit.

Lines 240 to 260 invite you to enter a new tax rate. If you do and press ENTER, all the figures in the table that use the tax rate are recalculated. This instant recalculation facility is the principle behind a type of financial planning program called a spreadsheet. Interrelated columns of figures representing income, raw material/production costs, overheads and so on can be entered. Then the effects of changing one or more of these parameters can be observed as all the totals are recalculated throughout the display:

TAX TABLE DISPLAY



ITEM	COST	No	SUB	TAX	TOTAL
1	1.98	20	39.6	3.17	42.77
2	2.4	19	45.6	3.65	49.25
3	5.6	11	61.6	4.93	66.53
4	1.05	45	47.25	3.78	51.03
5	4.35	15	65.25	5.22	70.47
6	2.99	15	44.85	3.59	48.44
7	1.92	24	46.08	3.69	49.77
8	7.2	4	28.8	2.39	31.19
9	5.45	6	32.7	2.62	35.32

Try a new tax rate

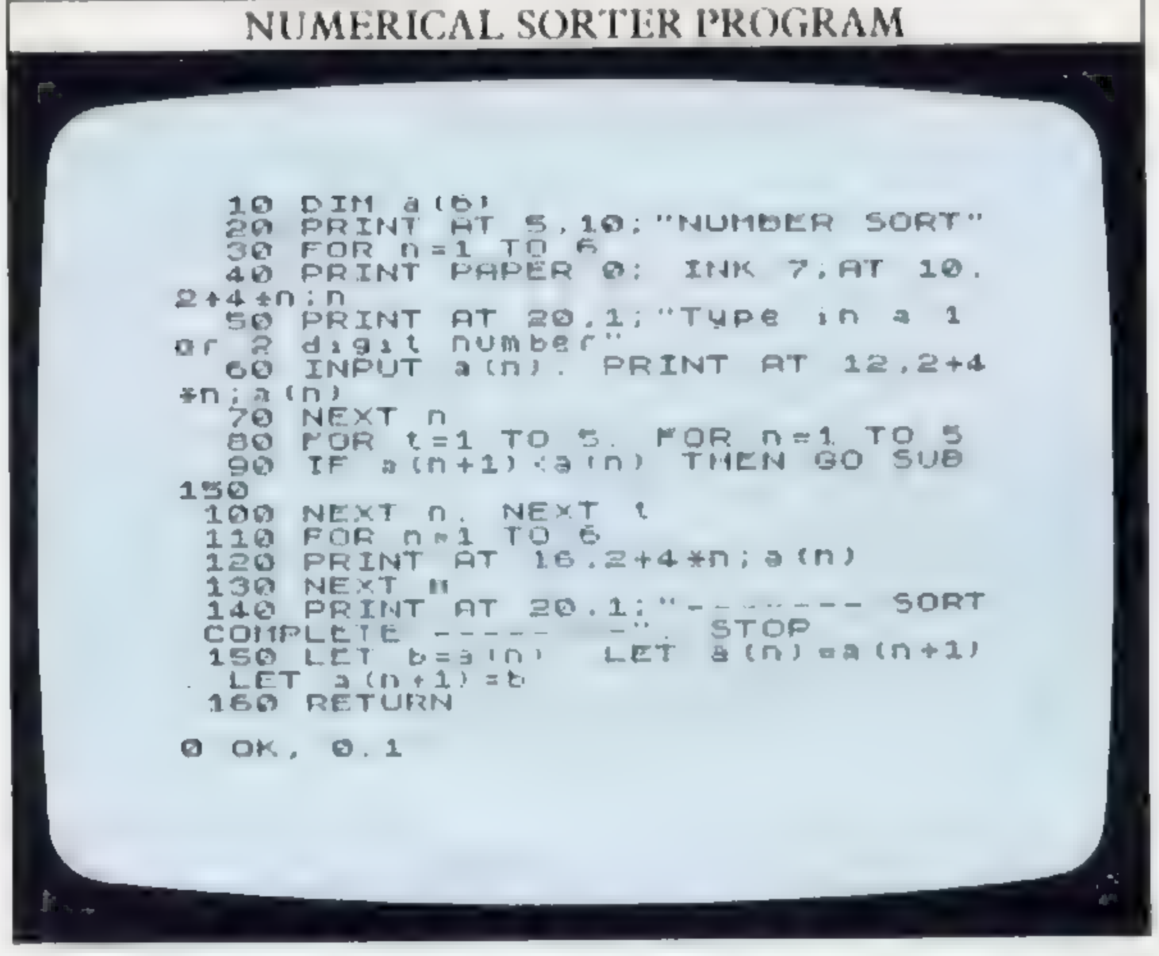
WORKING WITH WORDS 1

Almost every program you have looked at so far has taken numerical data and performed some calculations to arrive at a result. But the strings used have almost invariably been left in their original order. However, as you saw in the reaction test program on page 13, the computer stores letters as well as numbers as ASCII codes (a table of the Spectrum's ASCII codes appears on page 60). Because these codes have numerical values, the computer can examine strings and then reorder them in a similar way to numbers.

How to rearrange numbers

To see how words – or any strings – are sorted or reordered, it's helpful if you understand how the same thing is done with numbers. Here is a program which asks you for six numbers, and then rearranges them in numerical order. You could easily extend it to deal with many more numbers:

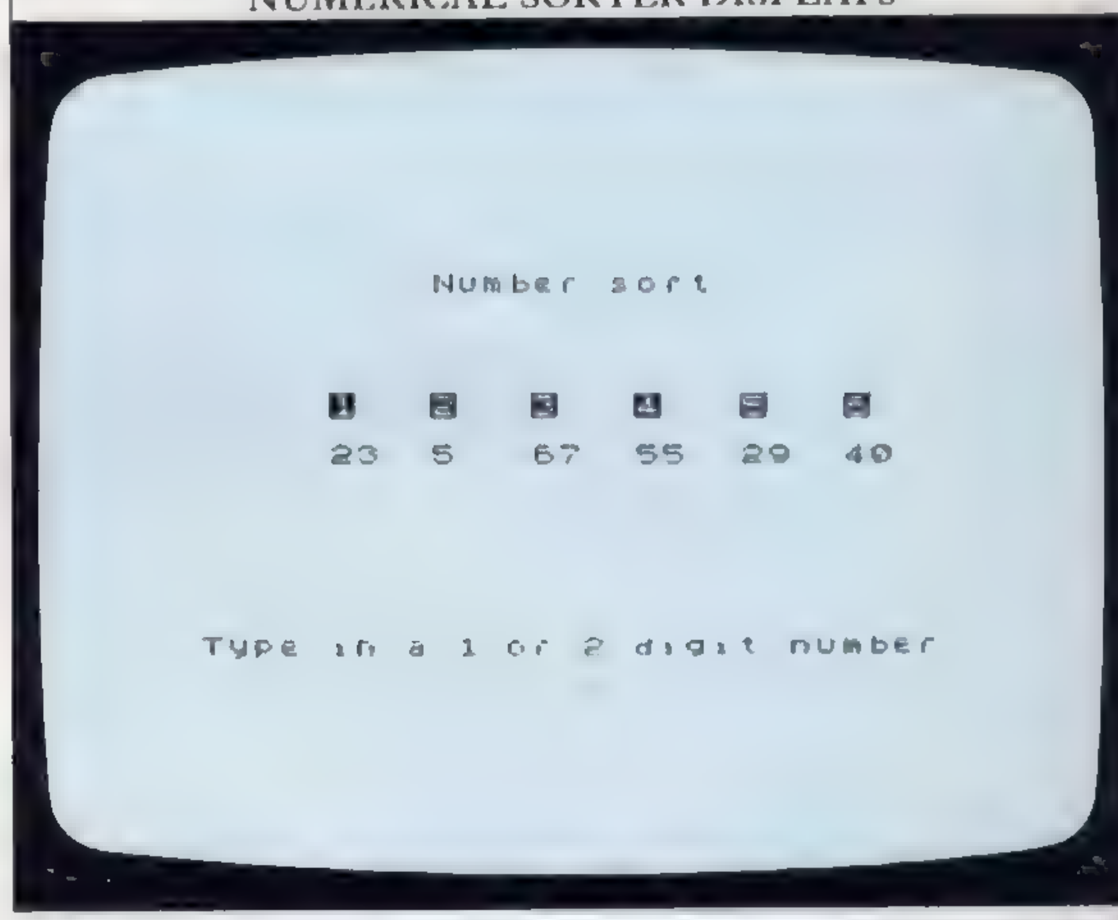
NUMERICAL SORTER PROGRAM

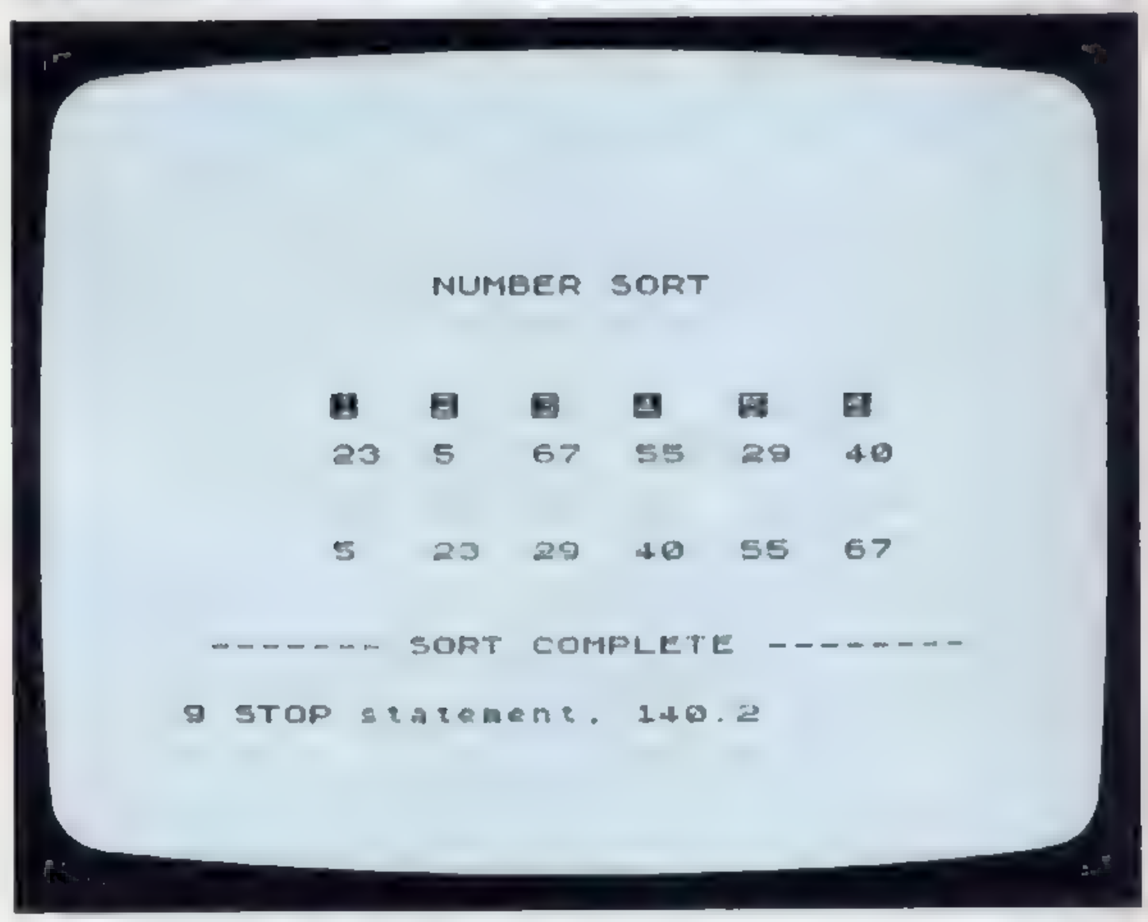


Line 50 asks you to type in a one- or two-digit number. Each time you do this and press ENTER, the next prompt (from 1 to 6) is PRINTed above the next INPUT position. This is where your chosen number will be PRINTed by line 60. All the numbers you ENTER are loaded into an array, so that they can be tagged with a number to identify them.

After you have ENTERed six numbers, the sorting procedure begins automatically at line 80. First look at the n loop (from the second statement in line 80 to line 100). For each value of n, a(n+1) is compared to a(n). If a(n+1) is the smaller, line 150 reverses them. So, on the first pass round the loop (n=1), if your first two numbers were 34 and 16, the reversal subroutine would be called, because the second number is smaller than the first. It would set b equal to a(1) – 34 in this case – set a(1) equal to a(2) – 16 here – and finally set a(2) equal to b, which on this loop is equal to 34. This juggling reverses the two numbers. The maximum number of sortings needed to put the list of numbers into the correct numerical order is five, so the t loop repeats the sorting process 5 times. These screens show the numbers before and after sorting:

NUMERICAL SORTER DISPLAYS





This reversal process can be used to manipulate strings in the same way. It's easy to see how the computer can compare two numbers and test whether the second is smaller than the first. We do it ourselves all the time – when comparing prices for example. But the computer can also decide whether "London" is less than "New York" – that is, whether one comes before the other in the alphabet. A line like:

```
50 IF "New York"<"London" THEN GOSUB 300
```

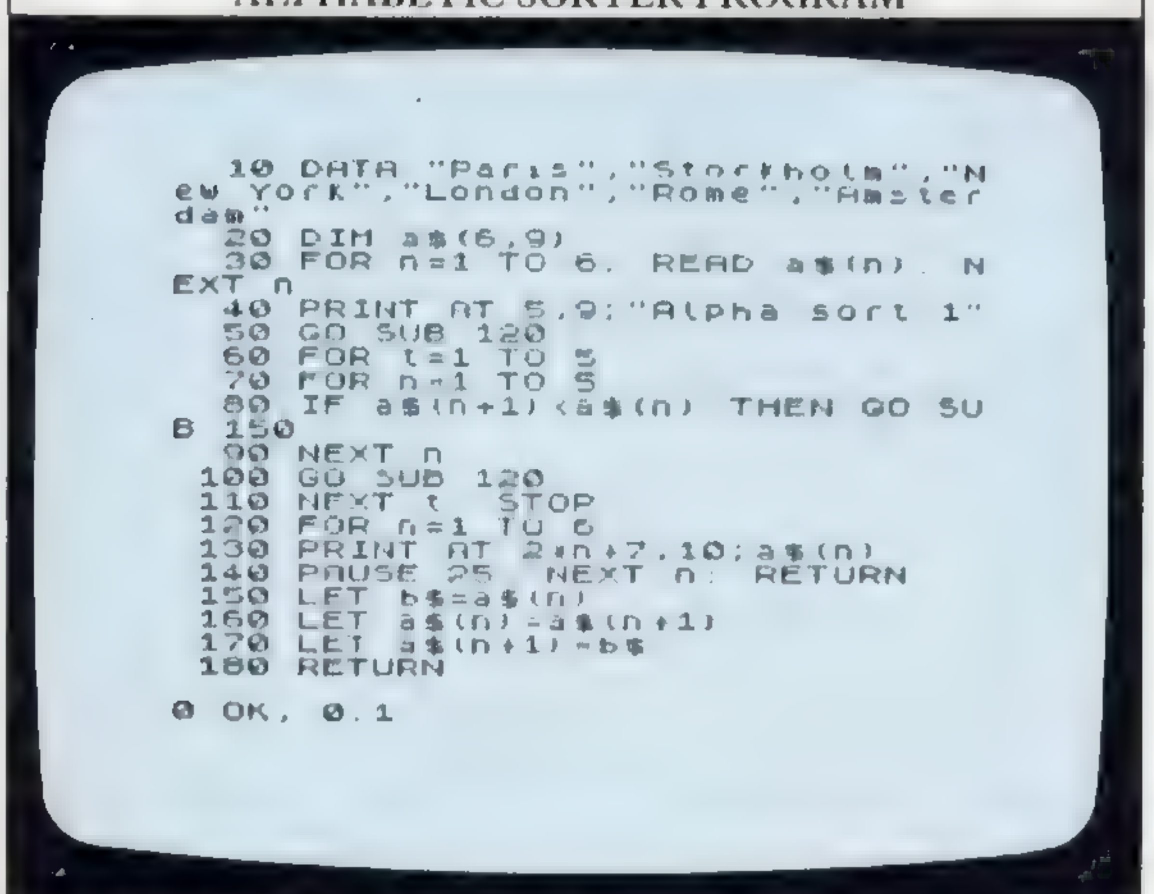
doesn't seem to make much sense at first glance. But because the computer stores a string like New York or

London as a series of numbers, these numbers can be compared and reordered. So comparisons like a<b and Stockholm>Paris both make perfectly good sense in BASIC.

Rearranging in alphabetic order

One of the most useful applications for string sorting is rearranging into alphabetic order. The following program shows one method – it works on strings already built into the program, but you can easily adapt the program to accept different strings by using INPUT:

ALPHABETIC SORTER PROGRAM



Line 20 dimensions a string array using the method demonstrated on page 22. Line 30 then READs the contents of line 10 – a series of capital cities. The subroutine at lines 120 to 140 PRINTs the cities in their original order. Then the sorting routine moves the strings around until they are in alphabetic order. You could add a PAUSE to slow things down so that you can actually follow what is going on:

ALPHABETIC SORTER DISPLAYS

The sorting routine at lines 150 to 180 is basically the same as that used to sort numbers. The variables used here are of course string variables, but the computer uses the same mathematical comparisons as it uses when operating with numbers. The program then STOPs at line 110.

Note that the temporary store used in both the number and string reordering programs (b and b$) need not be an element of an array. The variable is only used once on each sorting and then is not required again.

How to turn strings into numbers

The Spectrum uses a number of keywords which enable it to use numerical information from strings. In addition to taking string characters and comparing their ASCII values, the computer can decide how long a string is. To program this, you need to use the keyword LEN. The following program line for example produces a number:

```
100 n=LEN"Spectrum"
```

Here n is equal to 8, the LENgth of the string. You can use this in programs to reject an INPUT word or name that is too long, or make the computer take different courses of action depending on how long certain strings are. You can add LENs together to find out the length of a number of names or any other piece of text.

The Spectrum also has a keyword which operates on strings that are themselves numbers. VAL coverts a number-string containing a calculation into a number. The number produced by this is itself the result of the calculation. For example:

```
150 LET a$="2*34.5*0.3"
160 PRINT a$;"=";VAL a$
```

PRINTs both the calculation and its result – a useful programming technique. You can use this command to evaluate a string and then pass this value on to another part of a program.

WORKING WITH WORDS 2

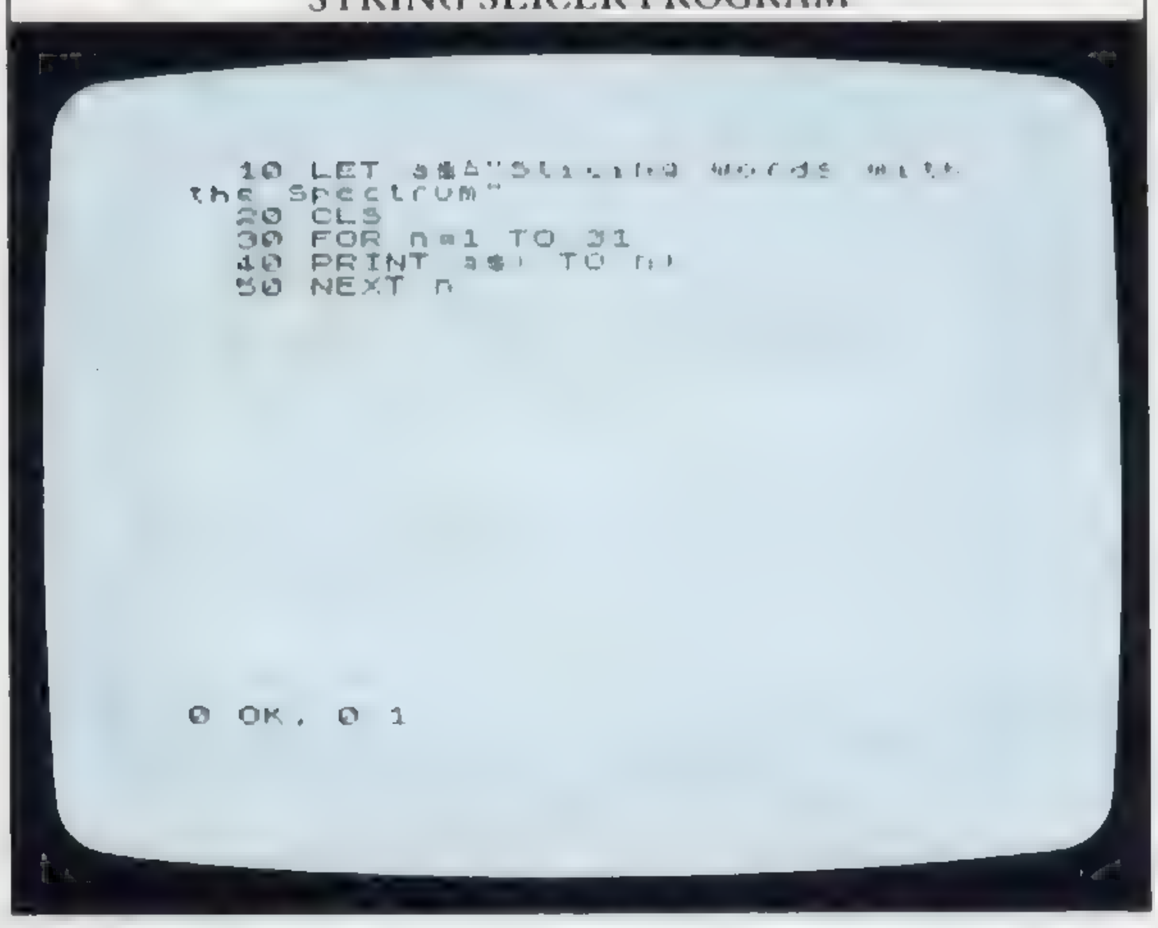
Until now, you have treated strings – or words that make up strings – as indivisible units. Some of the programs so far have added strings together, but none of them have "looked inside" the quotation marks that begin and end every string to work on the characters that are there. With the Spectrum you can take strings apart and reassemble their characters in a number of different ways. This means that you can program the computer to take out part of a word or group of words and examine it – a process that can be very useful.

Most computers that work with BASIC have a family of commands that can be used to manipulate strings – LEFT$, RIGHT$, MID$ and so on. They are used to pick out the first, last or middle character of a string respectively. Although the Spectrum doesn't have any of these commands on its keyboard, it can do everything that these commands do, if you know how to program it.

How to cut up words

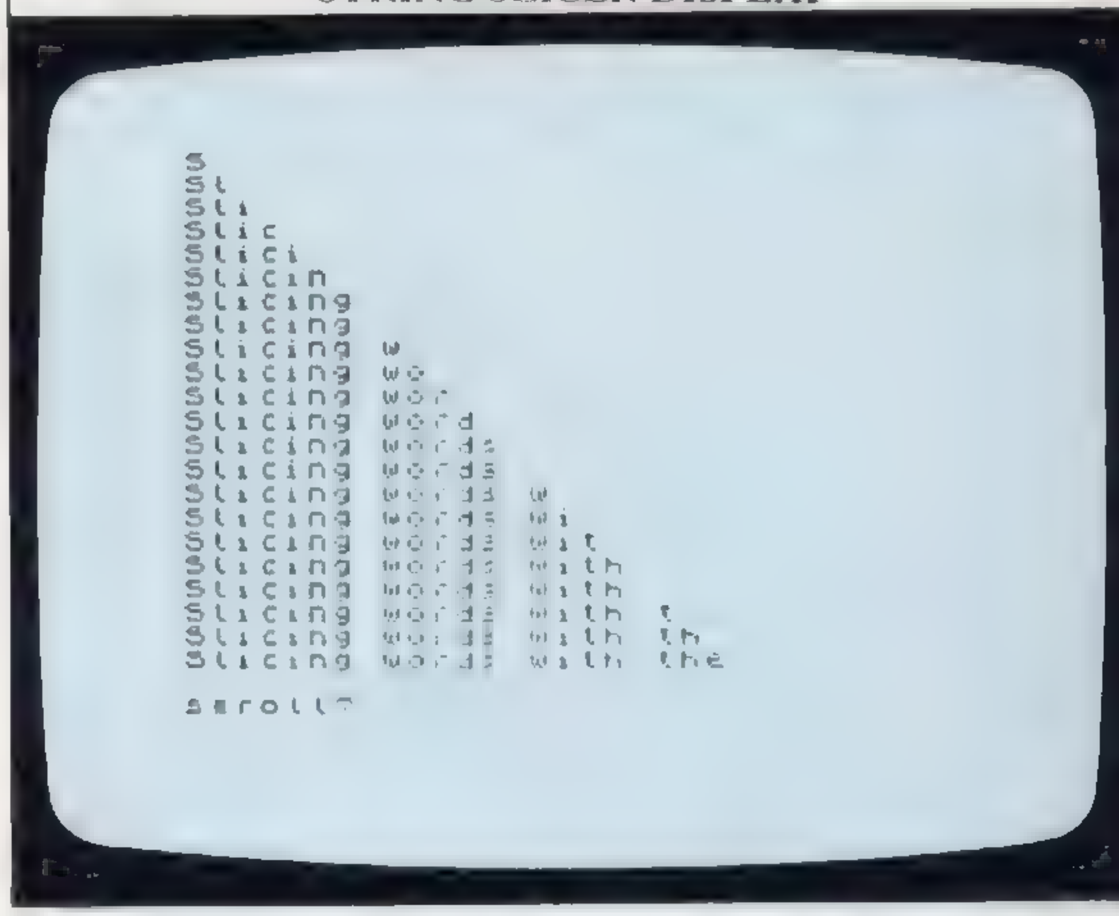
You can make the Spectrum break into a word by slicing parts off the string. It's fairly straightforward. To see how to do it, first type in the listing on the following screen:

STRING SLICER PROGRAM



The special technique here is in line 40, where a TO command appears as part of a string variable. For each value of n, line 40 PRINTs a string n characters long, from the first character to the nth character. So, the first line contains the string "S", the second line "Sl" and so on, until n equals the length that is set. With this program you can use any string – a group of words, numbers or other symbols; it's best if the value of n is not more than one screen line (32 characters). If you use a different string, make sure that the maximum value of n in line 30 is no more than the length of your string:

STRING SLICER DISPLAY



You can use this kind of technique to pick out strings that all begin with the same letter or word, and then perhaps PRINT them out in a series of lists.

The reverse effect is just as easy to produce. Try adding the following lines to the program, then RUN it to see what happens:

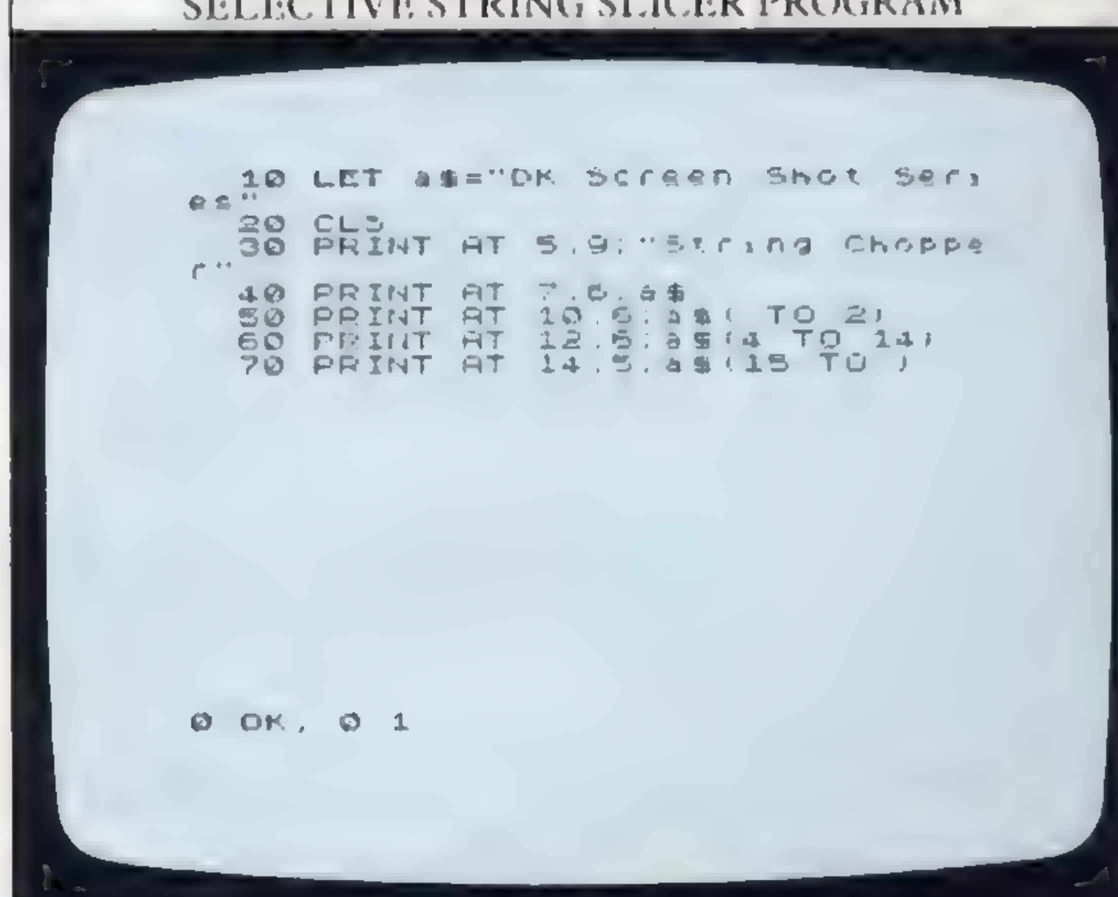
```
60 FOR n=1 TO 31
70 PRINT a$(TO 32−n)
80 NEXT n
```

Now, as n increases from 1 to 31 on each loop, the length of the string PRINTed decreases from 31 characters to only 1.

Picking out parts of a phrase

Now you can explore this technique. Type in and RUN the listing on the following screen:

SELECTIVE STRING SLICER PROGRAM

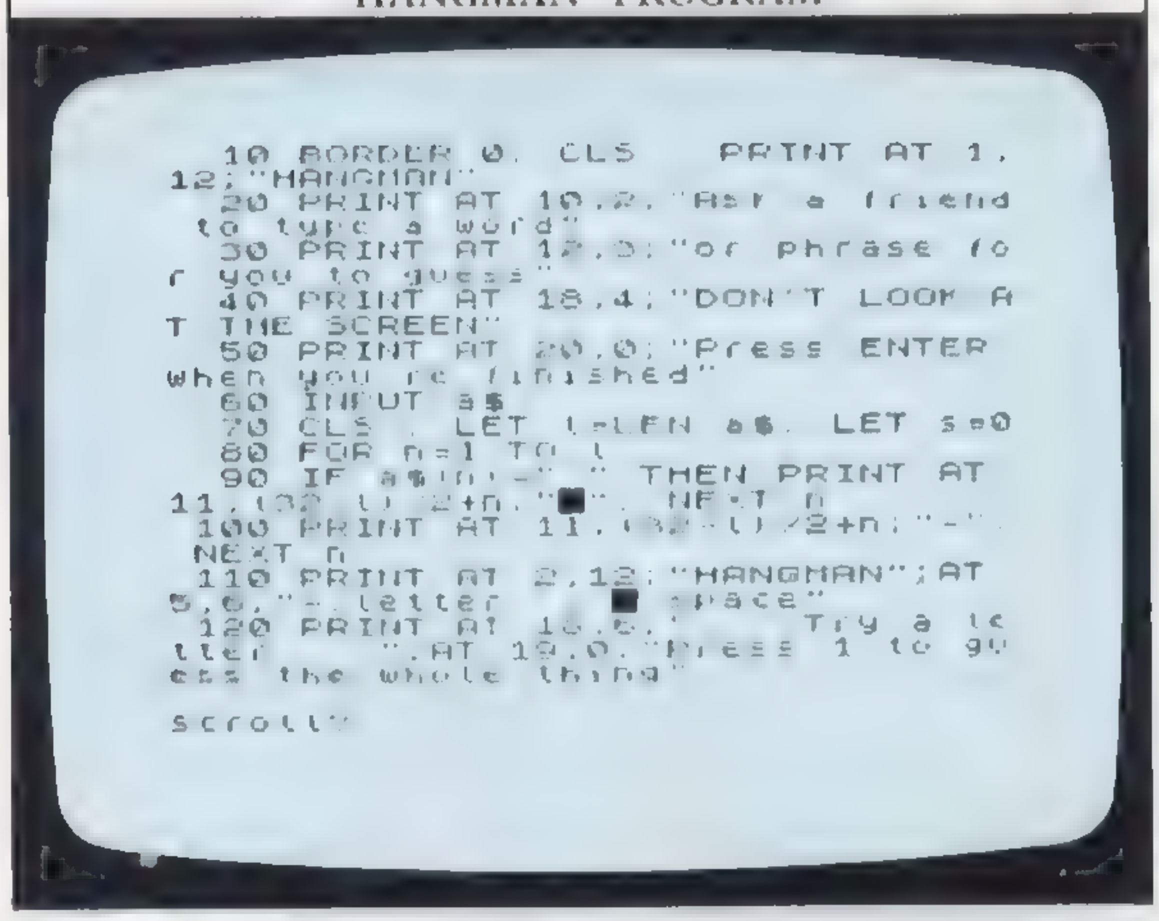


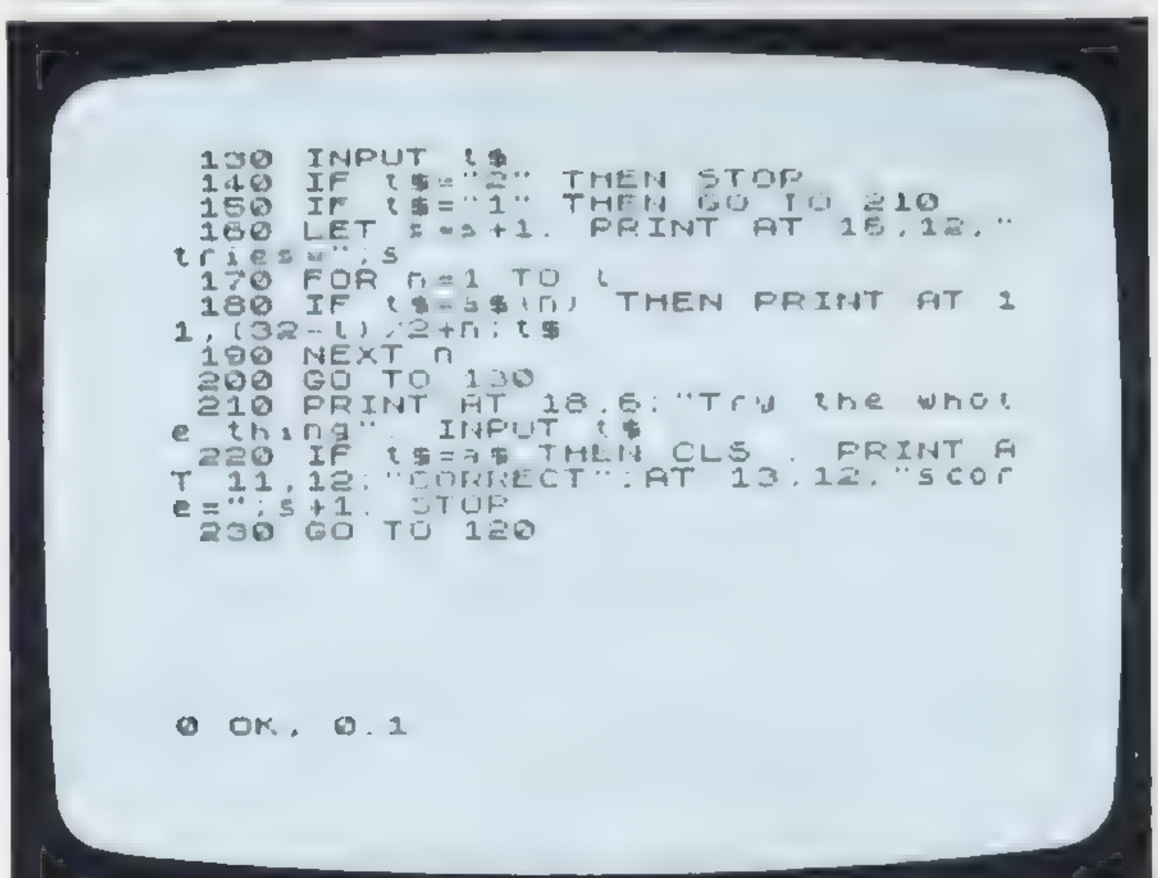
You aren't limited to dealing with the first n characters of a string. In fact, you can take any consecutive group of characters from a word or sentence. In this program line 50 works in the same way as line 40 of the slicing program. Line 60 forms a string from characters 4 to 14 out of the middle of a$. Finally, line 70 forms a third string from character number 15 to the end of a$. Although these three "substrings" are formed from parts of a$, a$ itself is still intact. This technique lets you take a group of words and pick out any of them for use on their own in a program.

Word games with string commands

The next program shows how you can use these methods of handling words in a game. It's a computerized word-guessing contest in which one player enters a word and the other has to guess it; the computer PRINTs the letters that the user has guessed correctly in their right positions in the word:

"HANGMAN" PROGRAM



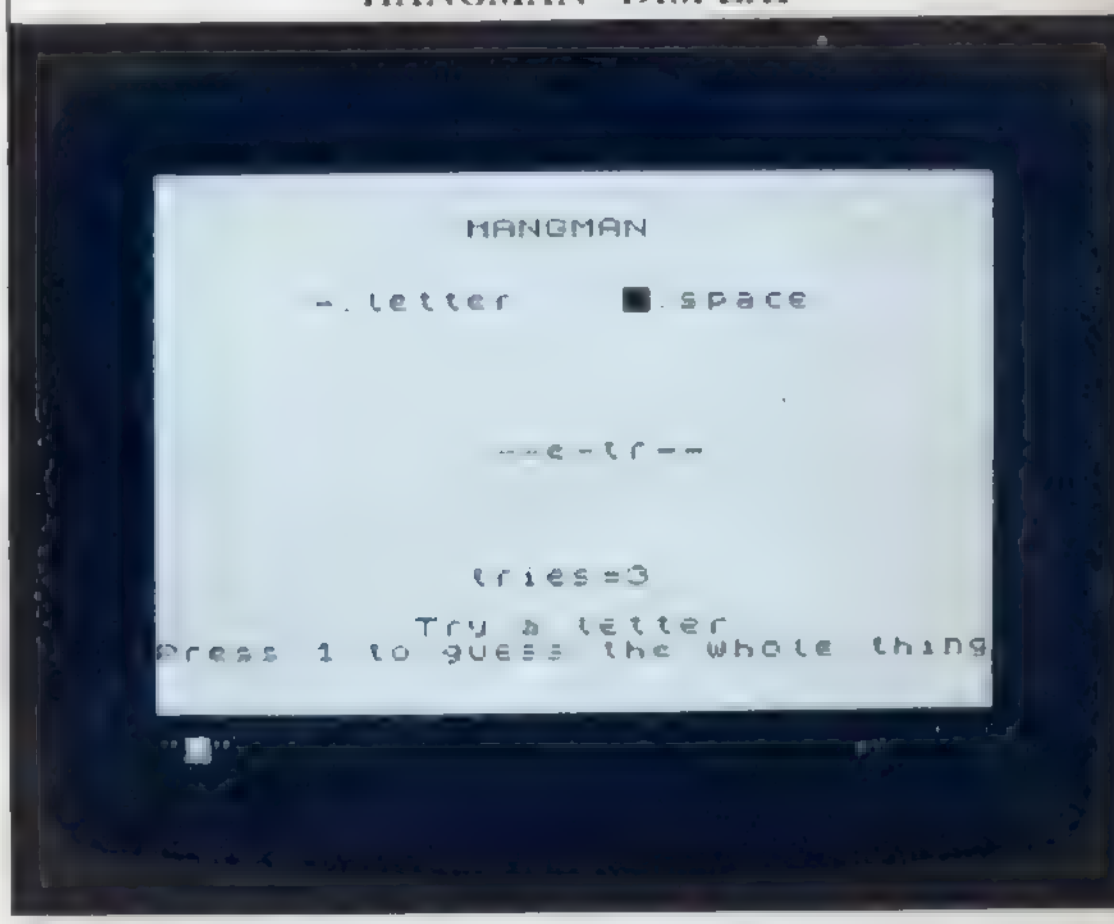


Lines 10 to 50 PRINT the title frame. When a friend has typed in the test string that you will have to guess, line 70 calculates the length of this test string using the command LEN, and sets the score, s, to zero.

The program now has to PRINT symbols on the screen to represent the letters in the test string. As you guess the letters, any correctly guessed letters will replace these symbols. Also, to allow for test phrases rather than just words, the positions of the spaces between the words are shown. Line 100 PRINTs hyphens to represent the characters. Line 90 PRINTs black squares to represent any spaces in the test string. In lines 90 and 100, (32−l)/2+n works out where the characters that represent the test string should be PRINTed so that they lie in the middle of the screen, not off to either side (a similar effect is incorporated in commercial word-processing programs).

If you want to guess the whole word or phrase instead of keying in individual letters (you can do this at any point in the game), press 1. The program jumps to line 210. The word or phrase that you type in (t$) is compared to the stored string (a$). Then a "CORRECT" frame is PRINTed or if the guess is wrong, the program returns to single letter entry. When a single letter is tried, lines 170 to 190 compare it to each character of the stored string in turn. If the guess is correct, the letter is PRINTed in the appropriate position in the display. Here is an example of the display you should see when a game is in progress:

"HANGMAN" DISPLAY



You can easily limit the number of guesses by adding the commands:

```
IF s>n THEN STOP
```

after the statements where the score, s, is calculated. If you should make a mistake when using the program, it can be difficult to interrupt using the shifted BREAK key, so to make this easier, add one extra line:

```
145 IF t$="3" THEN STOP
```

To stop the program at any point, simply press 3.

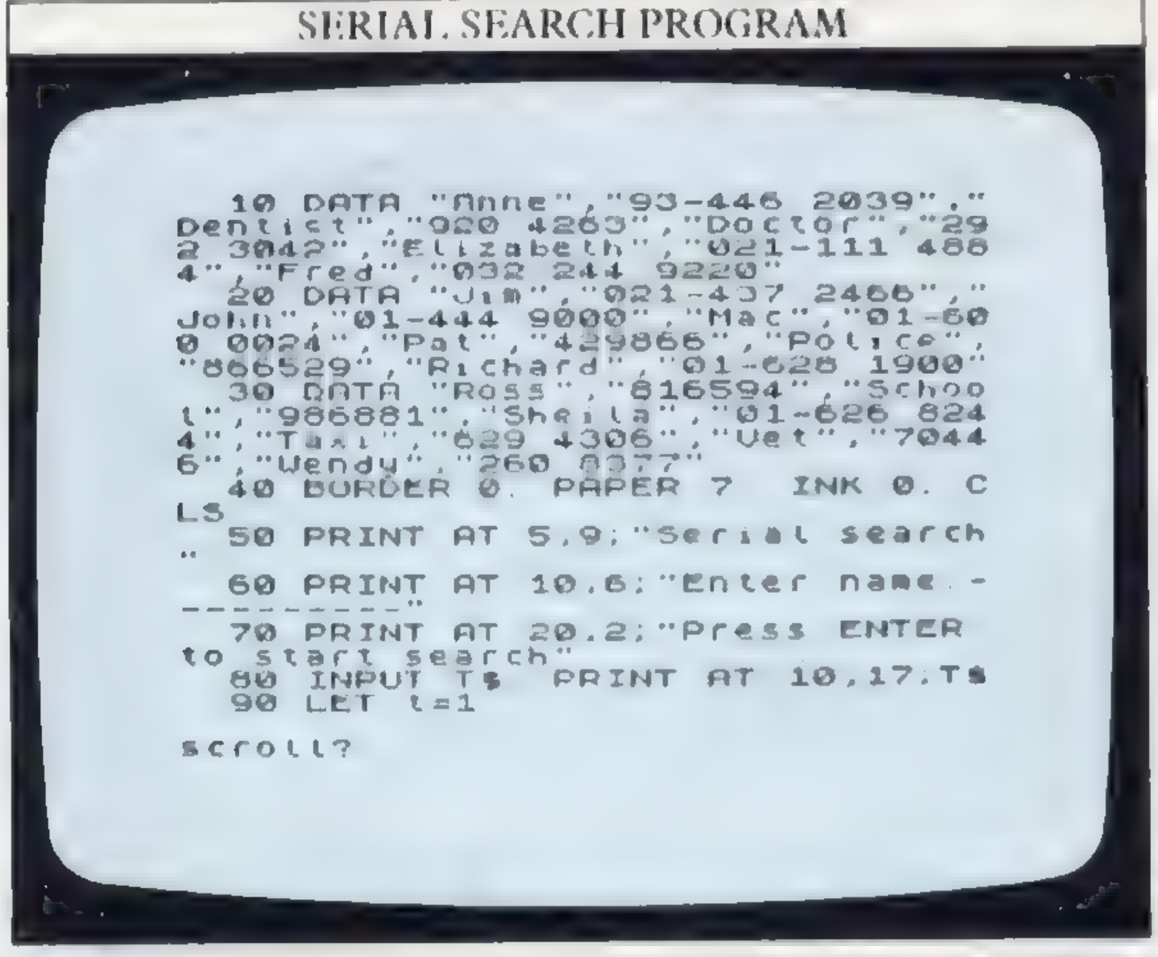
FACT-FINDING

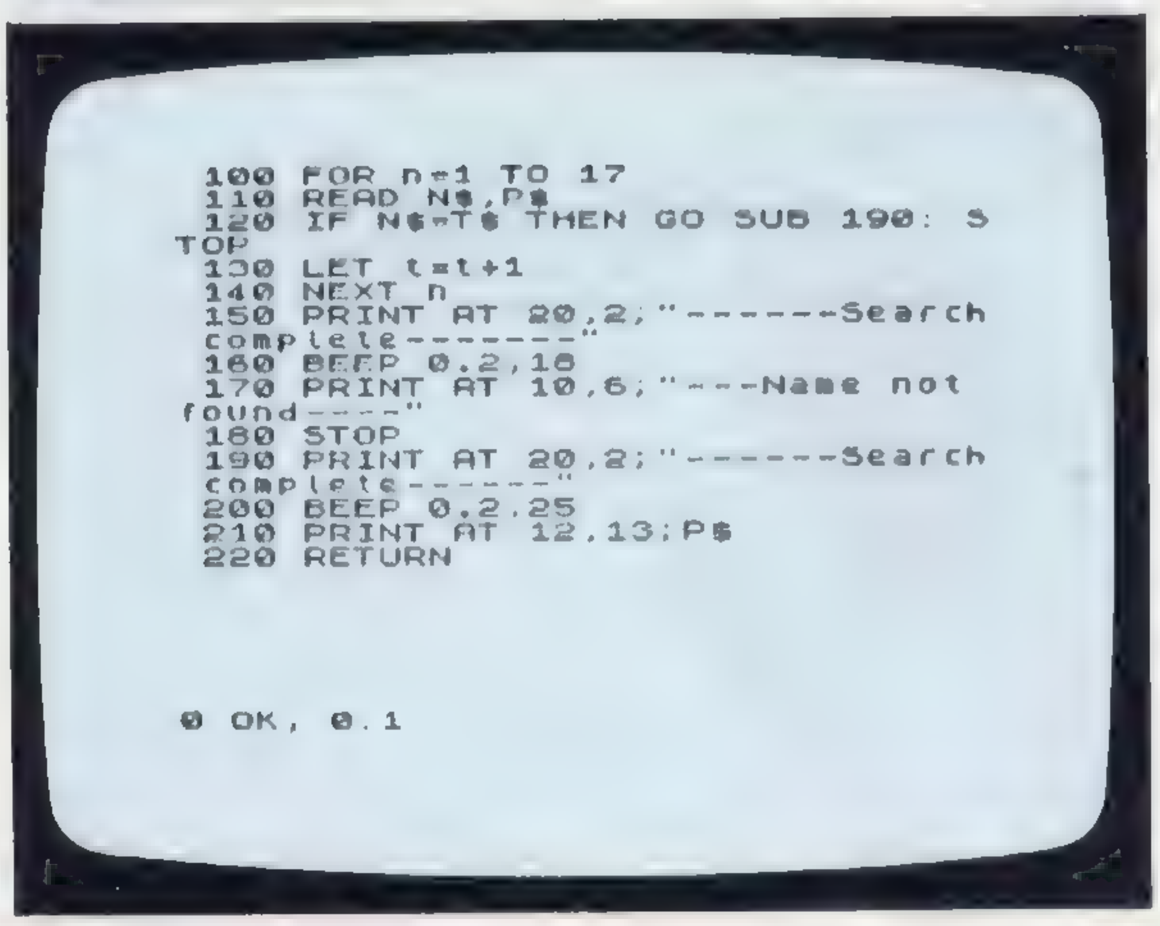
You can use your Spectrum to store information, rather like an electronic filing cabinet. However, it would be time-consuming if you had to display the complete contents of a long file every time you wanted to look up a single item. You would still have to scan the screen visually to find the piece of information you were looking for, as if you were using paper files. A good program lets you pick out information selectively, so that the computer, and not you, does the searching.

How to program a serial search

The program that follows uses a simple but effective method to locate one item in a long list of DATA. It's called a "serial search" because it searches the DATA in a series of stages. It takes the test string (T$) that you type in and compares it to each string in the program's DATA statements in turn. It carries on until the test string matches one of the stored strings:

SERIAL SEARCH PROGRAM





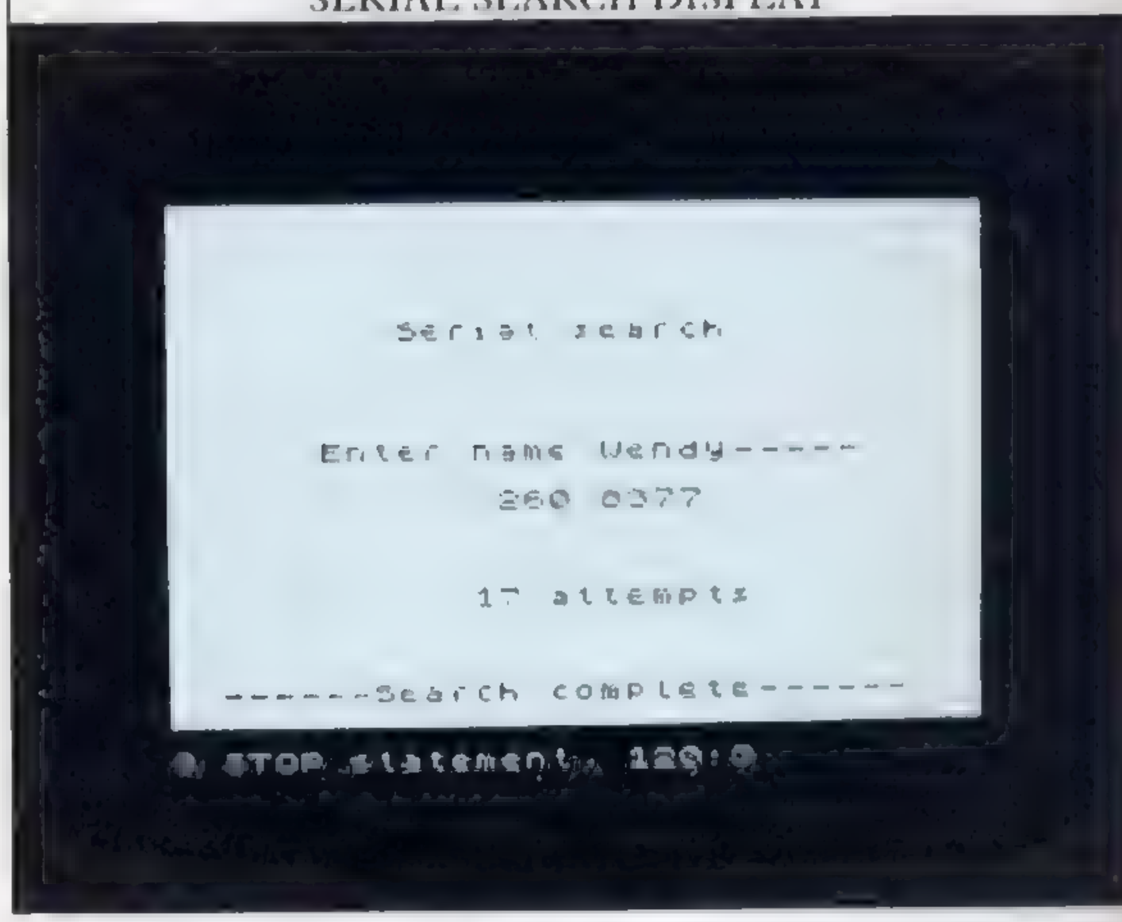
Lines 10 to 30 hold a list of names and telephone numbers. Lines 100 to 140 repeatedly READ a name (N$) and number (P$) from the DATA statements (note that the numbers are treated as strings). This carries on until a stored name is found to match the test string. Line 210 then PRINTs the phone number.

If the computer cannot match the test string with any string in the DATA lines, line 170 announces that the name has not been found. If you now add the following line you can count the number of searches made before a successful match is found:

```
215 PRINT AT 16,12;t;" attempts"
```

Now try RUNning the program to find the last telephone number on the list:

SERIAL SEARCH DISPLAY



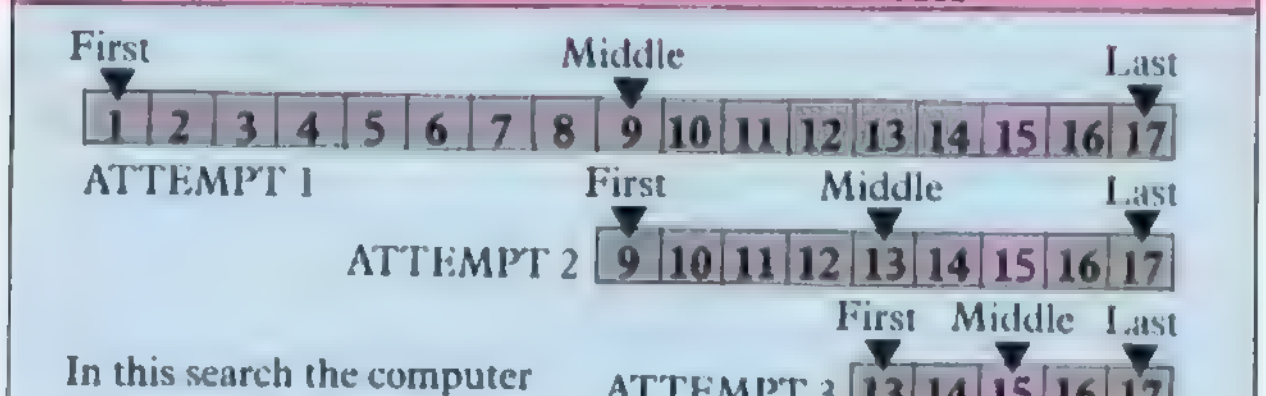
You will find that the program takes 17 searches and a fraction of a second to find the last name in the list. That's acceptable in a short program like this with just a few pieces of stored DATA. But for a serial search program with more than about 200 names or other items of DATA, the delay caused by looking at every single piece of DATA to find the correct one becomes quite noticeable. The program is simple but slow.

Speeding up a DATA search

Fortunately, another DATA searching method can speed things up considerably. It relies on the storage of DATA in strictly alphabetical or numerical order. The DATA stored in the program is repeatedly divided in two and the first item found there is READ and compared with the test string. This produces three possibilities. First, the two strings may be identical – a successful match at the first attempt. Second, the test string may come later in the alphabet or numerical order than the item found. Third, the test string may

come before the item found. If the match is not successful, the half of the DATA in which the test string lies is further divided in two and searched in the same way until the strings are matched.

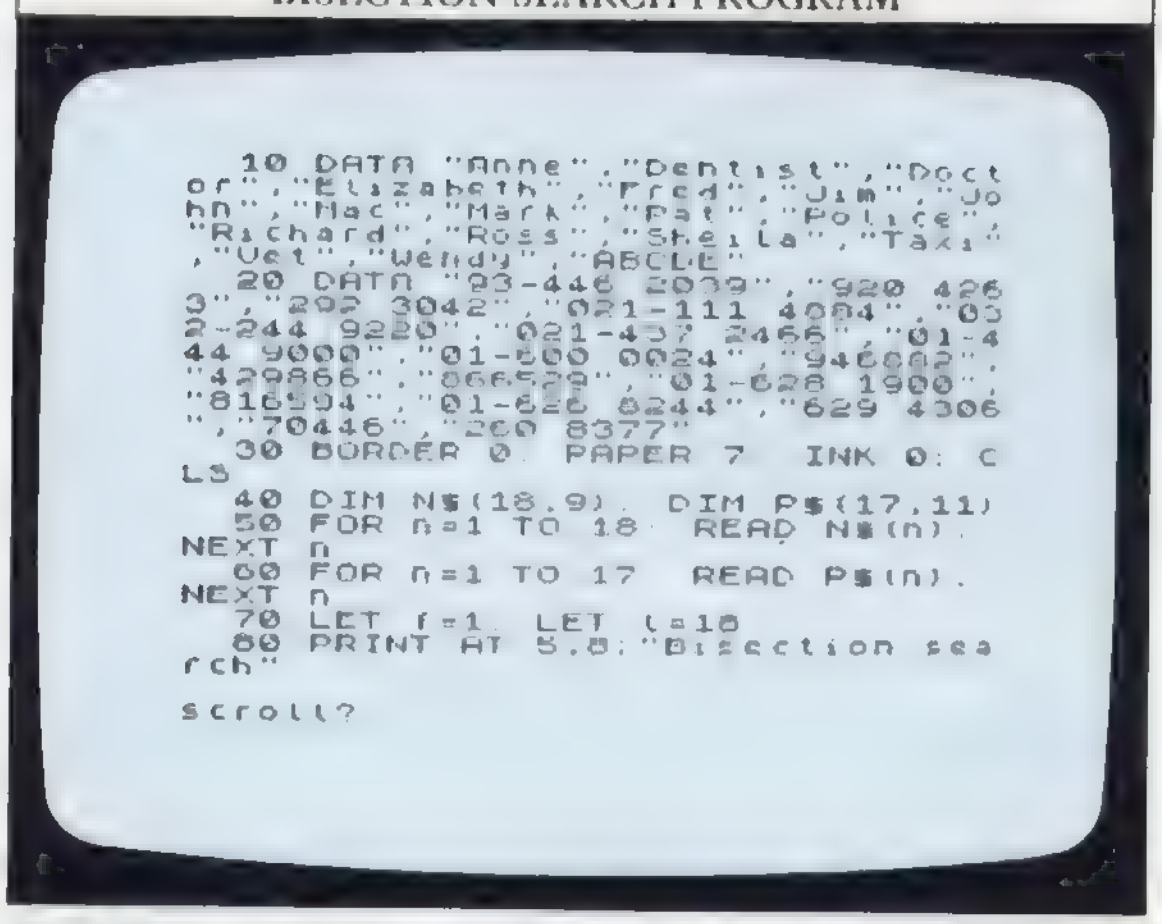
STAGES IN A BISECTION SEARCH

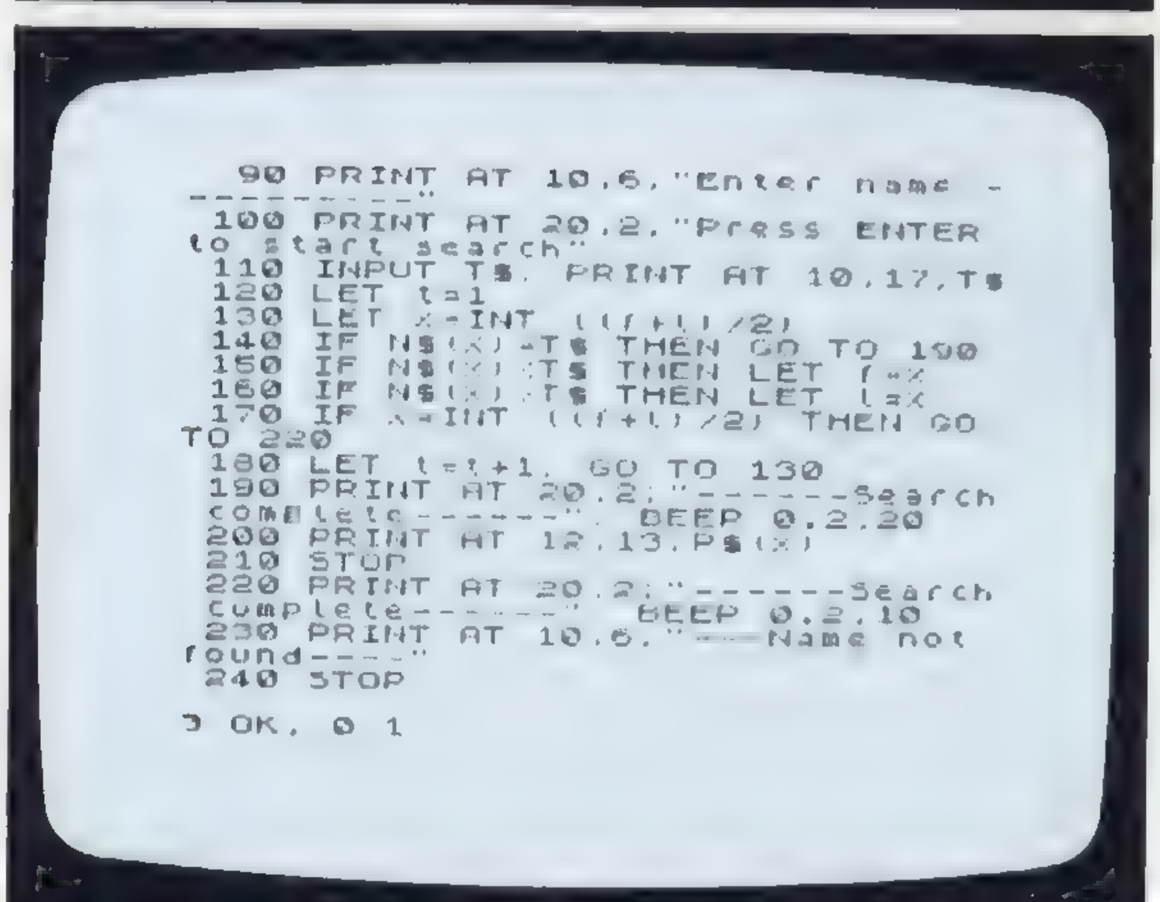


In this search the computer is looking for item 15.

Here is a program which carries out this "bisection search" of a DATA bank:

BISECTION SEARCH PROGRAM





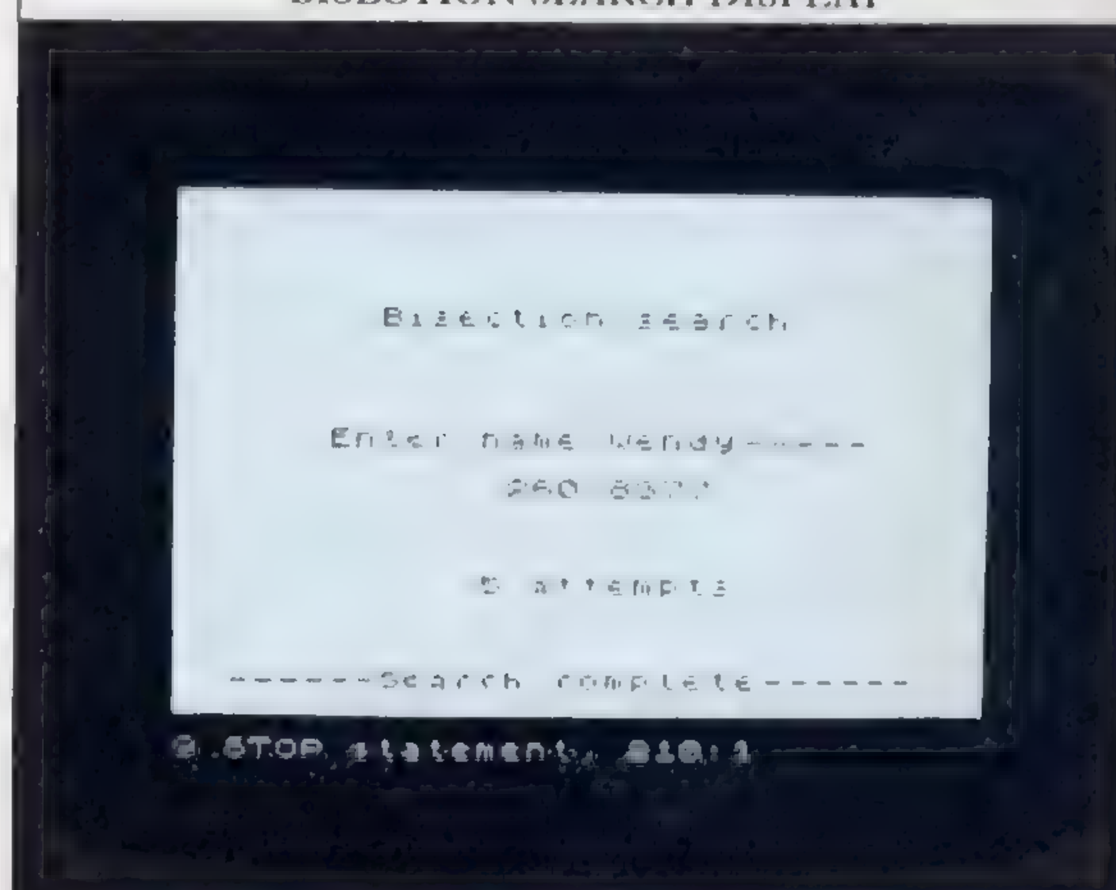
The DATA has been reorganized into two separate listings – one of names and a second of telephone numbers. In order to be able to locate one item in these DATA statements without having to READ through every item, the DATA must be numbered or tagged in some way. Line 40 does this by creating two numbered lists of DATA, or arrays. Lines 50 and 60 READ the relevant pieces of DATA into the two arrays. Line 130 sets the conditions for the first search. The first item in the range to be halved, f, is one and the last item, l, is 18. If a match is not made at line 140, lines 150 and 160 set the new values of f and l for the next search.

If you RUN the program in the form shown here, it will answer "Name not found" to every test string. That's because line 40 tells the computer that every name string (N$) is 9 characters long and every number string (P$) is 11 characters long. A name like "Wendy" is therefore stored as "Wendy   " which, as far as the computer is concerned, is not equal to the test string. To deal with this add:

```
111 LET T=LEN T$
112 IF T<>9 THEN LET T$=T$+" ":GOTO 111
205 PRINT AT 16,12;t;" attempts"
```

This keeps adding spaces to the test string until it is 9 characters long. Now the program RUNs and PRINTs the number of attempts to find a match:

BISECTION SEARCH DISPLAY



The last item in line 10 is not a misprint. Because of the way the program divides the DATA into halves, the last name (WENDY in this case) would actually never be located. The maximum value of x is one less than the number of DATA items. To get around this, the dummy item ABCDE is added, so that all of the DATA can be searched successfully.

Using this method, the last name can be found after only 5 attempts. To save time, the phone number, P$, is only READ when a successful match for the name is found. This time saving makes the bisection technique much more suitable if you want to search long lists of DATA. You could also use programs like those on pages 24–25 to order your DATA before you use it in the bisection search program. By combining the programs you would have an accessible DATA bank.

PIE CHARTS

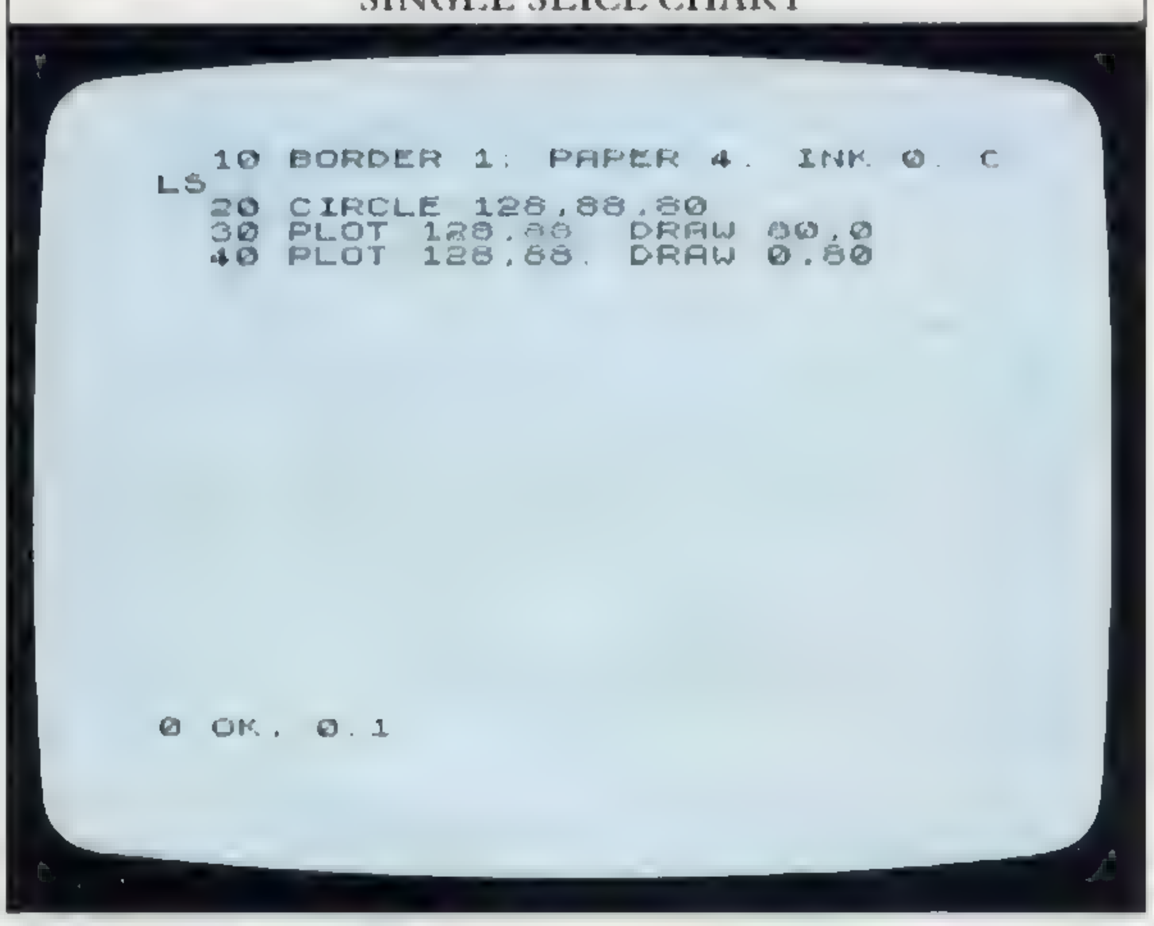
Computer graphics are invaluable for displaying information in a way that can be understood at a glance, and one of the most easily understood displays that computers can produce is a pie chart. Pie charts are particularly useful for showing the relationship between a quantity and the amount of which it is a part.

Drawing a fixed pie chart

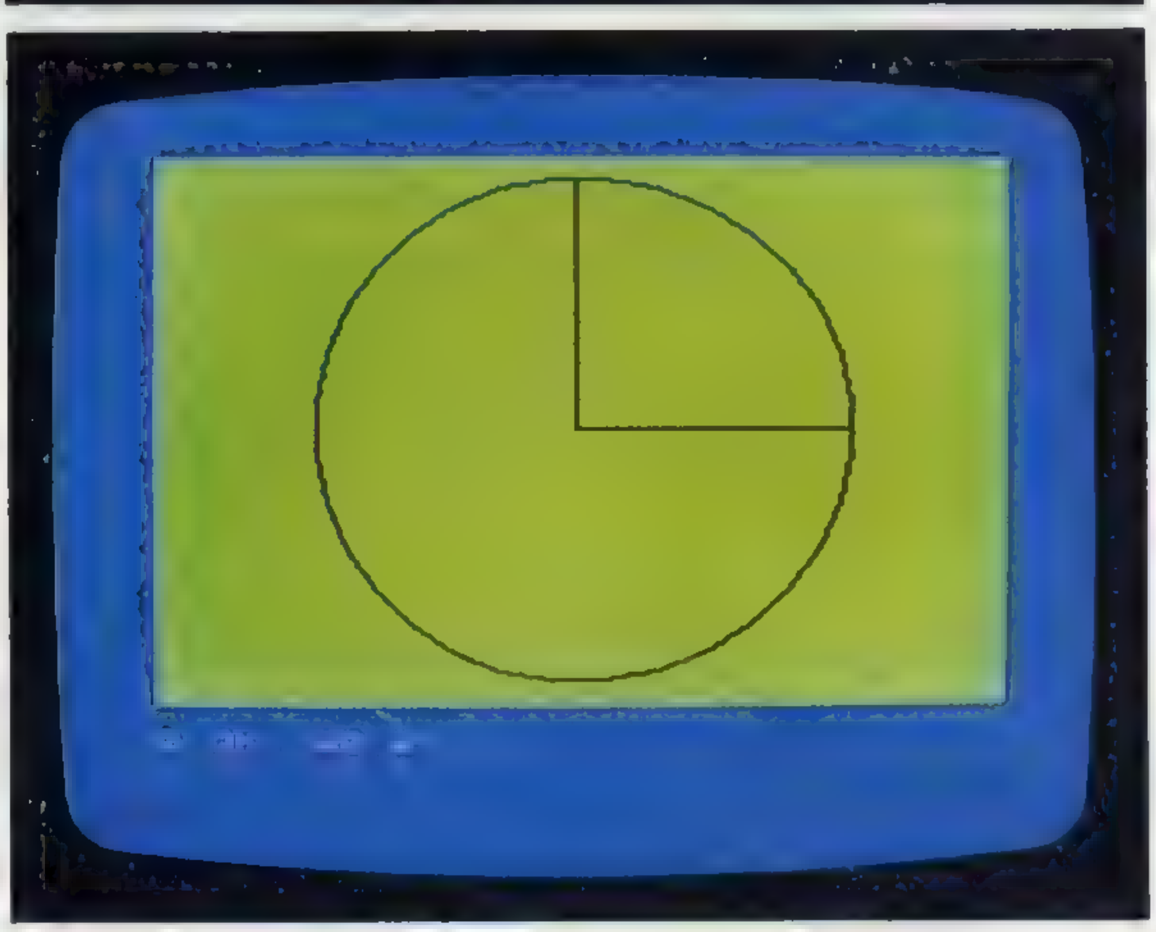
To produce a pie chart, the first job is to draw a circle, then you can start to put in the edges that mark off the "slices". Each edge is a radius of the circle. Once the first radius is DRAWn, all the other radii can be DRAWn relative to the first one. It's like cutting a pie; it doesn't matter where the first cut is made, but after that, the size of each slice is determined by the angle the slice makes with the previous one.

In the following program, one right-angled slice is DRAWn in a circle:

SINGLE SLICE CHART



```
  10 BORDER 1: PAPER 4. INK 0. C
LS
  20 CIRCLE 128,88,80
  30 PLOT 128,88  DRAW 80,0
  40 PLOT 128,88. DRAW 0.80

0 OK, 0.1
```



The program constructs the circle at the centre of the screen (128,88). Line 30 DRAWs the first radius from the centre to the right. Line 40 then DRAWs the second radius straight up to form the slice.

Adding more slices

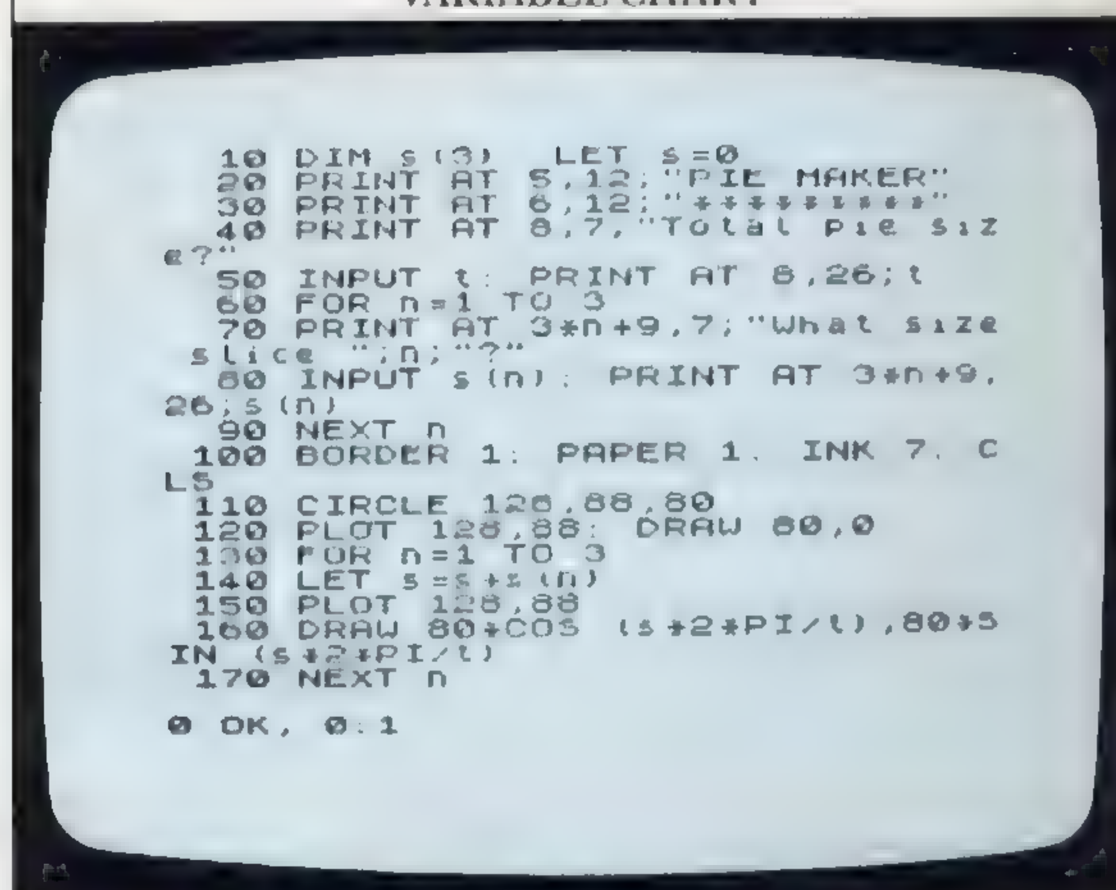
You could go on to DRAW a second quarter-circle slice by adding the line:

```
50 PLOT 128,88:DRAW –80,0
```

and from there you could go on adding lines to DRAW further radii and build up a pie chart with a number of slices. But this sort of program isn't really very useful because you have to work out the positions of all the edges of the slices beforehand.

However, the principle behind the first program can be used to write another program which will work out the positions of all the radii for you:

VARIABLE CHART



```
  10 DIM s(3)  LET s=0
  20 PRINT AT 5,12;"PIE MAKER"
  30 PRINT AT 6,12;"*********"
  40 PRINT AT 8,7,"Total pie siz
e?"
  50 INPUT t: PRINT AT 8,26;t
  60 FOR n=1 TO 3
  70 PRINT AT 3*n+9,7;"What size
 slice ";n;"?"
  80 INPUT s(n): PRINT AT 3*n+9,
26;s(n)
  90 NEXT n
 100 BORDER 1: PAPER 1. INK 7. C
LS
 110 CIRCLE 128,88,80
 120 PLOT 128,88: DRAW 80,0
 130 FOR n=1 TO 3
 140 LET s=s+s(n)
 150 PLOT 128,88
 160 DRAW 80*COS (s*2*PI/t),80*S
IN (s*2*PI/t)
 170 NEXT n

0 OK, 0:1
```

Lines 20 to 90 set up the title and INPUT frame. Line 10 tells the computer that the program will use a one-dimensional numeric array called s which will have three items in it. Each of these items will be the size of one of three slices that the loop at lines 60 to 90 asks you to type in.

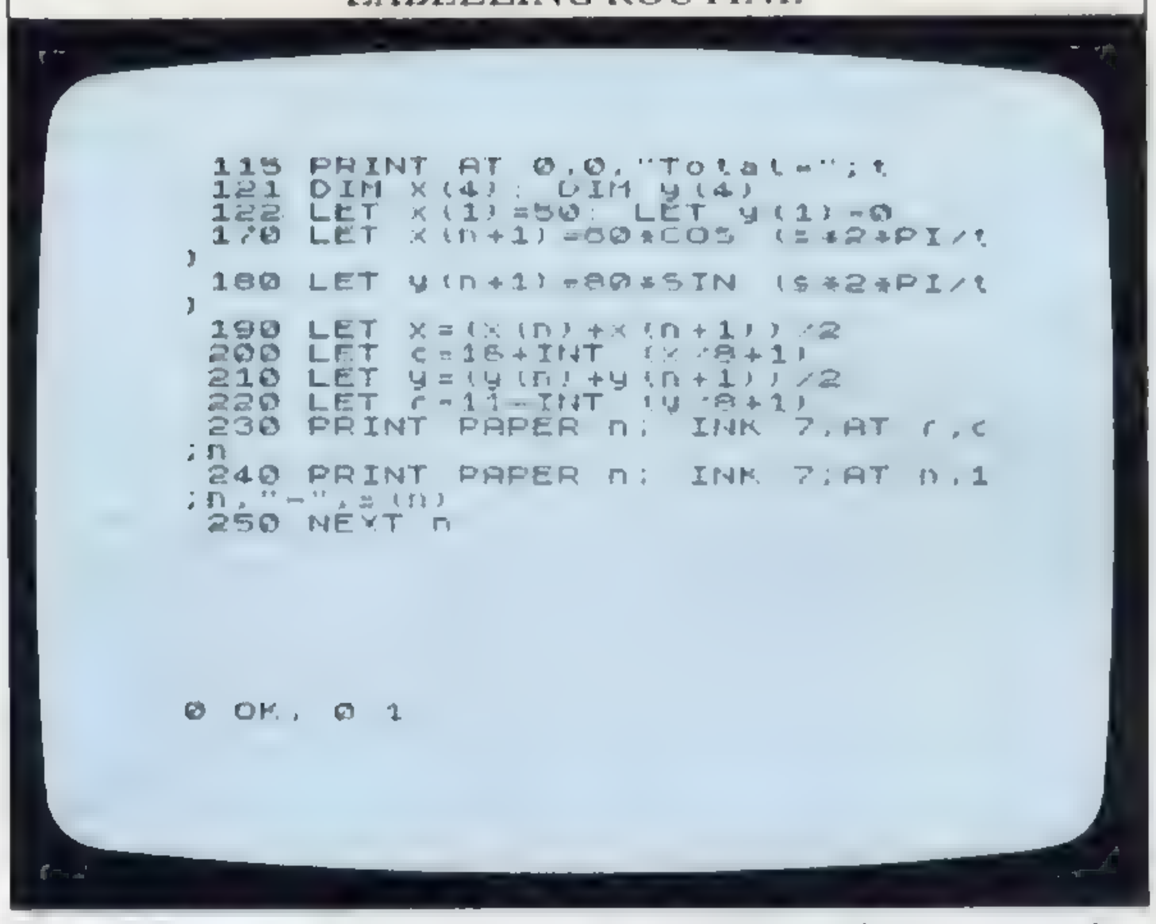
First, though, you must type in the total pie size in response to lines 40 and 50. Lines 100 to 170 DRAW the circle and radii in positions calculated using the slice sizes you typed in. The variable s, which was set to zero in line 10, fixes the position of each slice relative to the first radius. When n=1 for example, s=s+s(1). When n=2, then s=s+s(2). Finally, when n=3, s=s+s(3). For each value of s, a line is DRAWn from the circle's centre to the point calculated using COS and SIN in line 160.

You might wonder how this can DRAW lines which end to the left of, or below, the centre. When COS is used with angles between PI/2 and 3*PI/2 radians (90 and 270 degrees), the number it produces is negative. So if COS(s*2*PI/t) is equal to −1, the radius is DRAWn to an x co-ordinate of 128+(−80), or 48.

Labelling the slices

The previous program works well enough, but there's nothing to identify which slice is which – you need a good memory to know which slice represents which of the quantities you keyed in. So, next you can add a routine to label the slices:

LABELLING ROUTINE



```
115 PRINT AT 0,0,"Total=";t
121 DIM x(4): DIM y(4)
122 LET x(1)=50: LET y(1)=0
170 LET x(n+1)=80*COS (s*2*PI/t
)
180 LET y(n+1)=80*SIN (s*2*PI/t
)
190 LET x=(x(n)+x(n+1))/2
200 LET c=16+INT (x/8+1)
210 LET y=(y(n)+y(n+1))/2
220 LET r=11-INT (y/8+1)
230 PRINT PAPER n; INK 7,AT r,c
;n
240 PRINT PAPER n; INK 7;AT n,1
;n,"-",s(n)
250 NEXT n

0 OK, 0 1
```

Line 121 sets up two arrays, x and y, which are used to store the co-ordinates of the ends of the radii DRAWn on the circle. There are four pairs, not three – those for the first fixed radius plus three for the slices whose sizes you have typed in. The co-ordinates of the end of the first radius are known (50,0), and these are set in line 122.

For each value of n, the co-ordinates of the end of that radius are picked out of the arrays by x(n+1),y(n+1). The program uses these co-ordinates to PRINT a label on the screen. The position for the label is given by two co-ordinates x and y. These co-ordinates are half the distance between x(n) and x(n+1) respectively. The two graphics co-ordinates are then translated into text row and column numbers by lines 200 and 220.

The two co-ordinates x and y are each divided by 8 because there are 8 graphics pixels to each character position. But there is a disadvantage to calculating the label position this way. Since the label is to be PRINTed between the ends of the two radii, it will not work properly if the slice is more than half of the circle. However, for values less than this, the program works correctly and labels the pie chart for you:

LABELLED CHART

For each label, line 240 PRINTs the same number against a corresponding coloured background and PRINTs the size of the slice beside it to give you a display of the figures keyed in.

Try keying in the following answers in response to the question frame:

500 – total income
160 – bills
100 – insurance
180 – travel

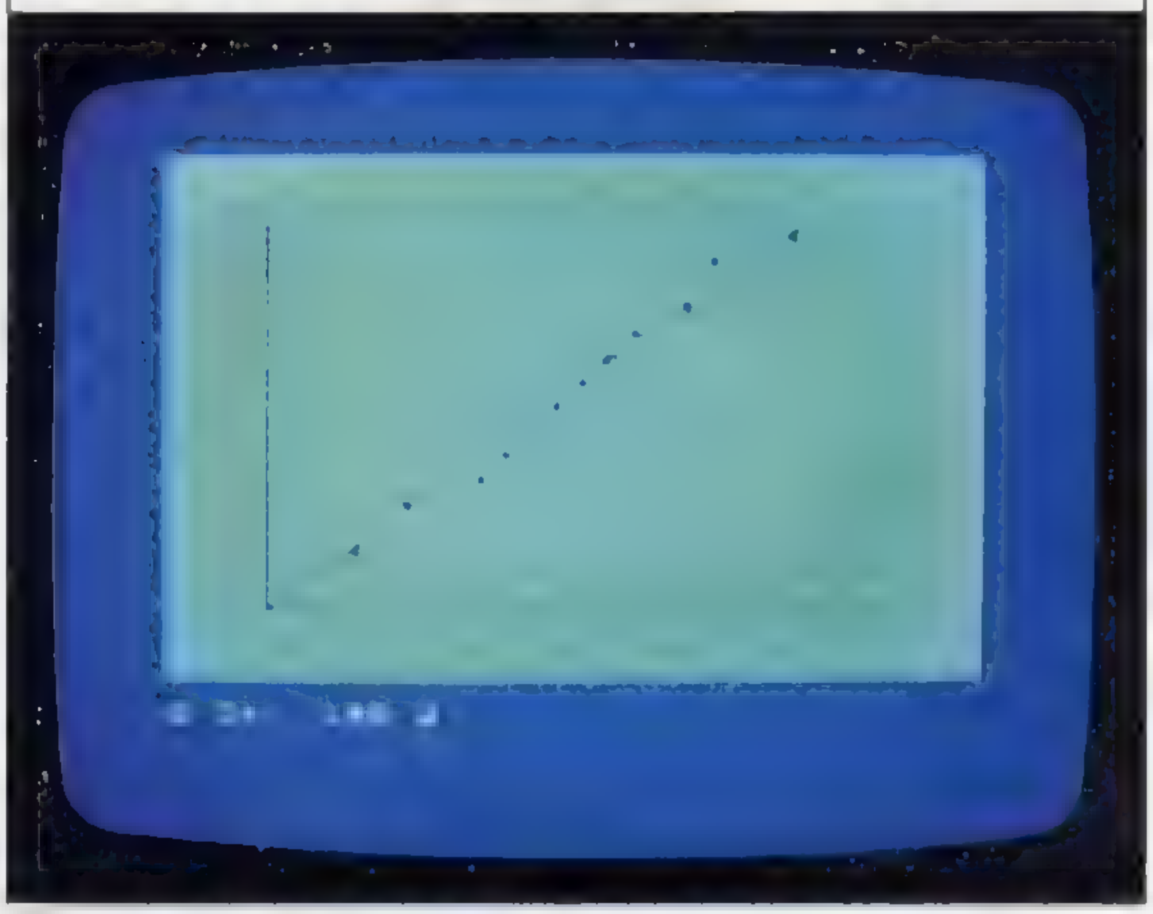
If you try these figures, you will end up with a chart that has a black unlabelled slice. This represents the amount of income that you have left over after the payments you have keyed in.

You can adapt this program to show more than three slices by changing the number of INPUTs and the dimensions of the arrays. It is possible to make the Spectrum DRAW sections of a circle that are filled in with colour. However, if you use routines that DRAW with INK to fill in sections on the pie chart, you will find that you have difficulty getting smooth edges to the sections. This is because the Spectrum's INK works at text rather than graphics resolution.

DRAWING GRAPHS

Because they are so good for games, computer graphics are often only partially exploited for more serious uses. But as you saw on the previous two pages, you can use your Spectrum to produce high-resolution graphics to show any information that you can put into number form. Pie charts are useful for showing how something is split up. Graphs, on the other hand, show how two sets of items are related. Here is a simple graph display:

FIXED GRAPH



You don't need to be a mathematician to get some useful information from this graph. As time goes by (along the horizontal axis), the value measured by the vertical axis is steadily increasing.

How to set up a graph

The program that produces the graph above has to DRAW the horizontal and vertical axes, label them, PLOT the points and finally DRAW a line:

FIXED GRAPH PROGRAM

```
  10 BORDER 1. PAPER 5. CLS
  20 DATA "A","B","C","D","E","F
"
  30 DATA 16,7,14,9,13,12,12,13,
10,15,9,16,8,17,7,18,6,20,4,21,3
,24
  40 PLOT 224,24. DRAW -192,0. D
RAW 0,128
  50 FOR c=4 TO 24 STEP 4
  60 READ m$
  70 PRINT INK 2.AT 20,c;m$
  80 NEXT c
  90 LET l=0
 100 FOR r=18 TO 2 STEP -1
 110 PRINT INK 2;AT r,2;l
 120 LET l=l+1
 130 NEXT r
 140 FOR n=1 TO 11
 150 READ r,c
 160 PRINT AT r,c,"*"
 170 NEXT n
 180 PLOT 32,24  DRAW 168,128

0 OK, 0.1
```

The sets of information are contained in the DATA in line 30. Line 40 DRAWs the two axes of the graph. The two loops that follow, between lines 50 and 80 and between lines 100 and 130, PRINT the labels along each axis. The first loop takes each letter in turn out of the DATA in line 20 and then PRINTs it along the horizontal axis. The position along the axis is determined by c in line 70. This increases in STEPs of 4 (line 50) so that there is a gap between each of the labels. Line 110 PRINTs numbers up the vertical axis. This time, because the labels are a sequence of numbers, there is no need for them to be stored in a DATA line. Instead, they are produced by the STEP in line 100.

Programming graphs to order

The main disadvantage of the previous program is that it will only ever produce the same graph. You can change the information in the DATA lines, but this is a laborious way of altering the display. Ideally, you want a program that will allow you to type any co-ordinates you wish while the program is RUNning. Also, you don't want to have to translate the graph's co-ordinates into Spectrum screen co-ordinates before using them. So, the program must be able to do this conversion for you.

The next program does all this. Although it produces axes that have set labels, you can key in any co-ordinates you like, as long as they fall within the graph's limits:

VARIABLE GRAPH PROGRAM

```
  10 BORDER 2  PAPER 6. INK 1. C
LS
  20 PLOT 224,24. DRAW -192,0. D
RAW 0,128
  30 FOR c=4 TO 28 STEP 4
  40 PRINT AT 19,c;c-4
  50 NEXT c
  60 LET n=0
  70 FOR r=18 TO 2 STEP -2
  80 PRINT AT r,2;n
  90 LET n=n+4
 100 NEXT r
 110 PRINT AT 0,2;"Y"
 120 PRINT AT 20,31;"X"
 130 INPUT "X=";h,"Y=";l
 140 BEEP 0.2,20
 150 PRINT AT 18-16*l/32,4+h;"*"
 160 GO TO 130



0 OK, 0 1
```

Line 20 DRAWs the axes as before. Lines 30 to 50 PRINT the x axis labels. Here the program produces labels that could be time in hours. The program again uses FOR ... NEXT loops to determine not only what the labels are to be, but also where they are to go. As the column numbers of the horizontal label positions go

from 4 to 28 and the values go from 0 to 24, the program uses the column number to produce the label in line 40.

Lines 60 to 100 PRINT the labels up the vertical axis. Here they are chosen so that they can represent a range of temperatures in Centigrade.

Line 130 invites you to enter a pair of co-ordinates. Type in a time, press ENTER and type in a temperature followed by ENTER again. The computer BEEPs and line 150 PRINTs an asterisk at the screen position:

VARIABLE GRAPH DISPLAY

How to alter the display

The graph display is quite coarse, because asterisks can only be PRINTed in the 384 character positions within the graph's area of 24 columns by 16 rows. For a more detailed graph, you could replace line 150 by:

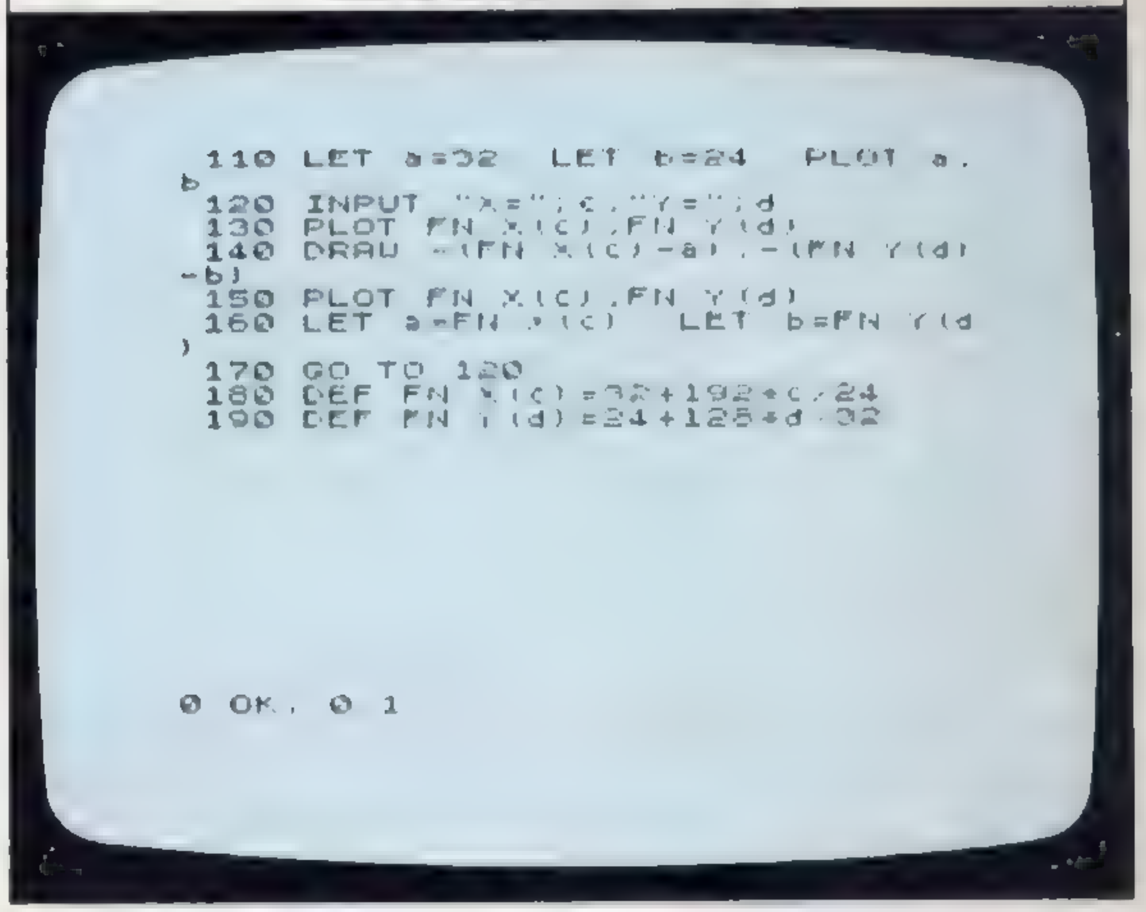
```
150 PLOT 32+192*h/24,24+128*t/32
```

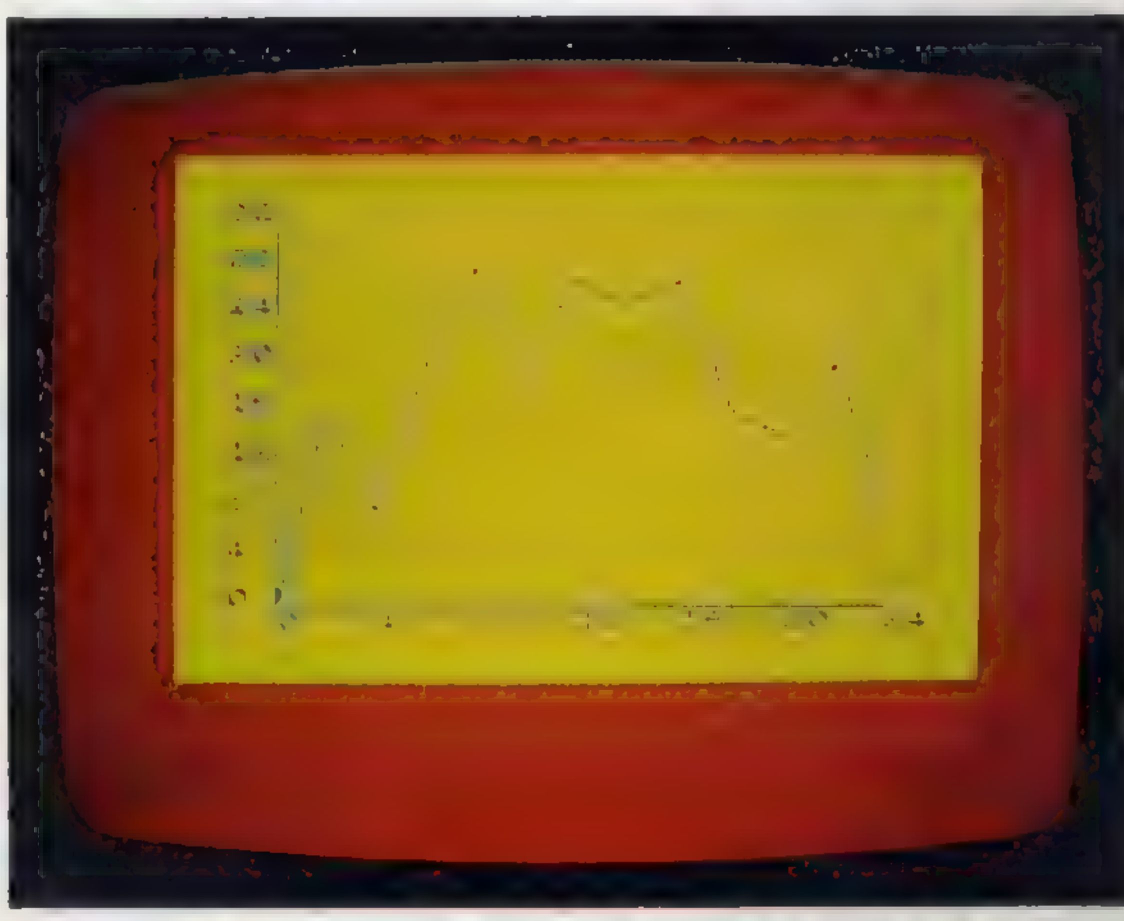
This will produce a display with points instead of asterisks, each positioned on a graphics grid that allows higher resolution:

GRAPH WITH POINTS

Altering the program so that it produces connected lines, rather than isolated points, is a little more difficult. What you need to do is tell the computer to PLOT a point, DRAW back from there to the previous point and then PLOT the second point again so that it in turn can be DRAWn back to. A program like this is ideal for writing with functions. Without them, the program lines would have to include repetition of a pair of quite cumbersome calculations for establishing co-ordinates. Here are the line changes and additions that are needed to make the program produce the line graph, together with a display:

LINE GRAPH CHANGES





Two functions are defined by lines 180 and 190. They convert horizontal and vertical graph co-ordinates into Spectrum screen co-ordinates so that the program can PLOT and DRAW with them. Line 110 sets the first point (a,b) at the bottom left corner of the graph. Then every time you INPUT a pair of co-ordinates, the program converts them, PLOTs a point at x,y, DRAWs back to a,b, PLOTs x,y again and lastly, at line 160, makes a,b equal to x,y.

BAR CHARTS

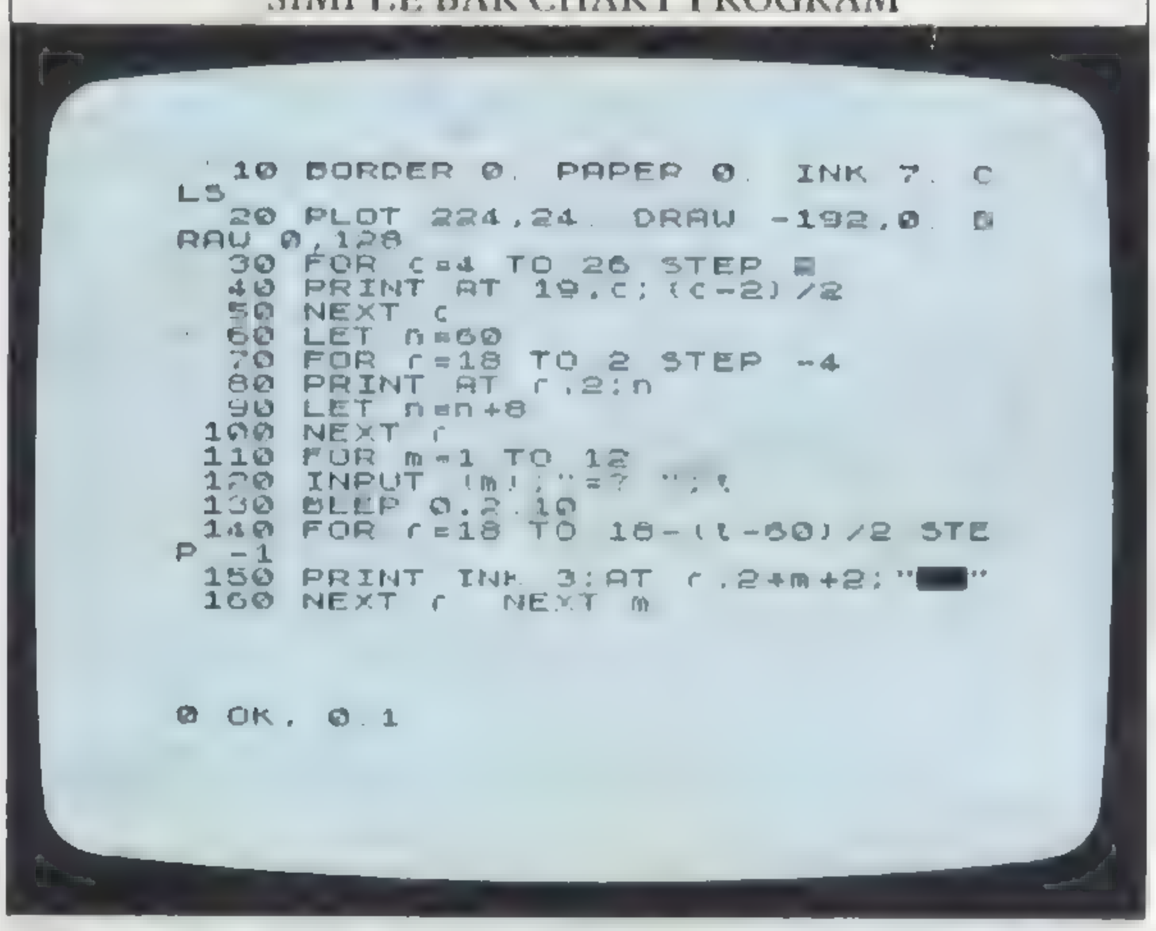
Graphic information can be presented in a number of different ways. Graphs, as you saw on the previous two pages, are good for showing trends, while bar charts are particularly useful for showing differences in levels.

Bar charts are so named because the information in them is displayed not as single points, but as columns whose height corresponds to the size or level of the item shown. You will frequently see bar charts used on television to show changes in currency exchange values, numbers of votes in elections and so on.

Writing a bar chart program

Because a bar chart is essentially a graph, you can use much the same techniques as used in a conventional graph program to produce one. The main difference is that instead of PLOTting a single point when fed with co-ordinates, the program must construct a column. With the Spectrum, these are most easily constructed from a series of square graphics characters (using the graphics cursor plus shifted key 8). INK can then be used to add colour to the chart. The next program incorporates this technique:

SIMPLE BAR CHART PROGRAM



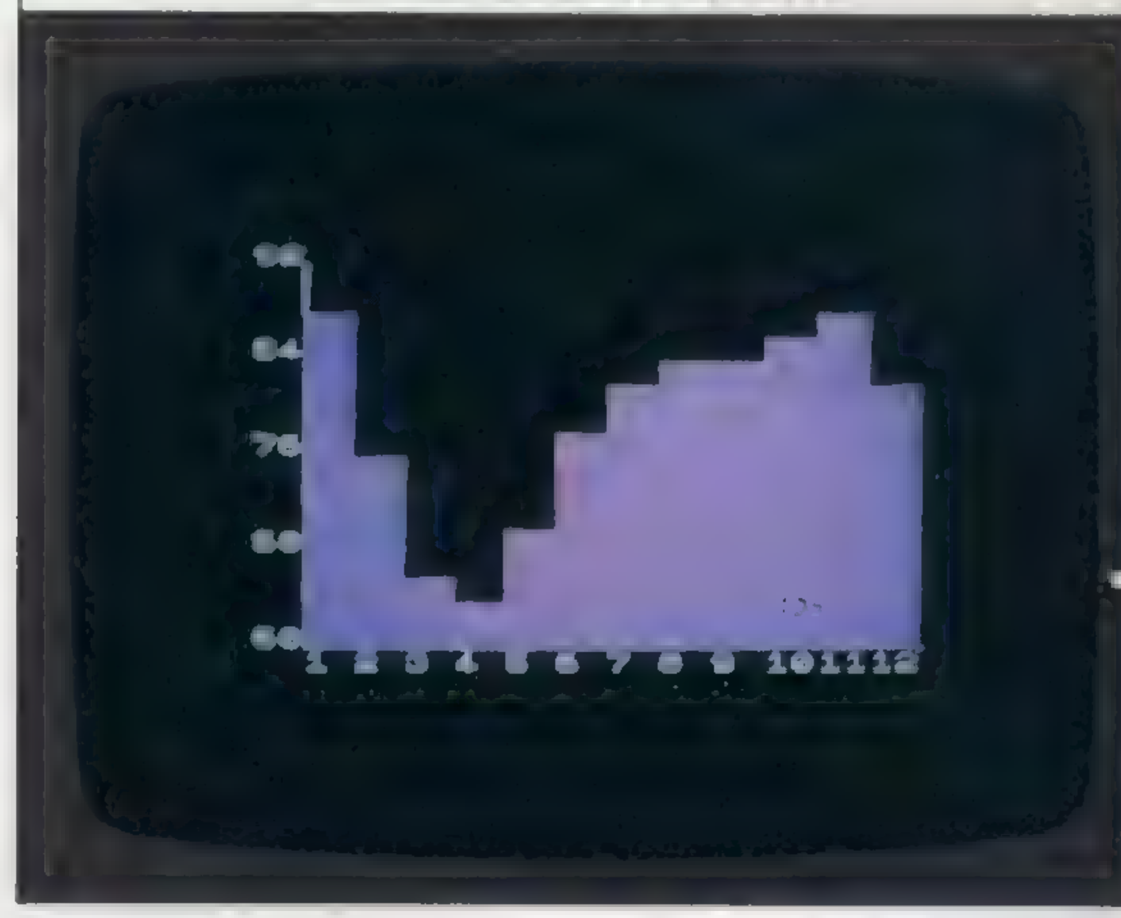
```
 10 BORDER 0: PAPER 0: INK 7: C
LS
 20 PLOT 224,24: DRAW -192,0: D
RAW 0,128
 30 FOR c=4 TO 26 STEP 2
 40 PRINT AT 19,c;(c-2)/2
 50 NEXT c
 60 LET n=60
 70 FOR r=18 TO 2 STEP -4
 80 PRINT AT r,2;n
 90 LET n=n+8
100 NEXT r
110 FOR m=1 TO 12
120 INPUT (m);"=? ";t
130 BEEP 0.2,10
140 FOR r=18 TO 18-(t-60)/2 STE
P -1
150 PRINT INK 3;AT r,2*m+2;"█"
160 NEXT r: NEXT m


0 OK, 0:1
```

The axes are DRAWn by line 20 in the same positions as those for the graphs on the previous pages. The FOR … NEXT loop at lines 30 to 50 labels the x axis with the numbers 1 to 12, which could represent the months of the year. If the columns that the labels are to be PRINTed under go from 4 to 26 (STEP 2) then the month number is given by:

month=(column−2)/2

Try it – column 4, at the beginning of the axis, is equivalent to month 1. Column 26, at the end of the axis, is equivalent to month 12. There are 12 STEPs altogether. Here is the program in action:

SINGLE-COLOUR CHARTS



Combining charts

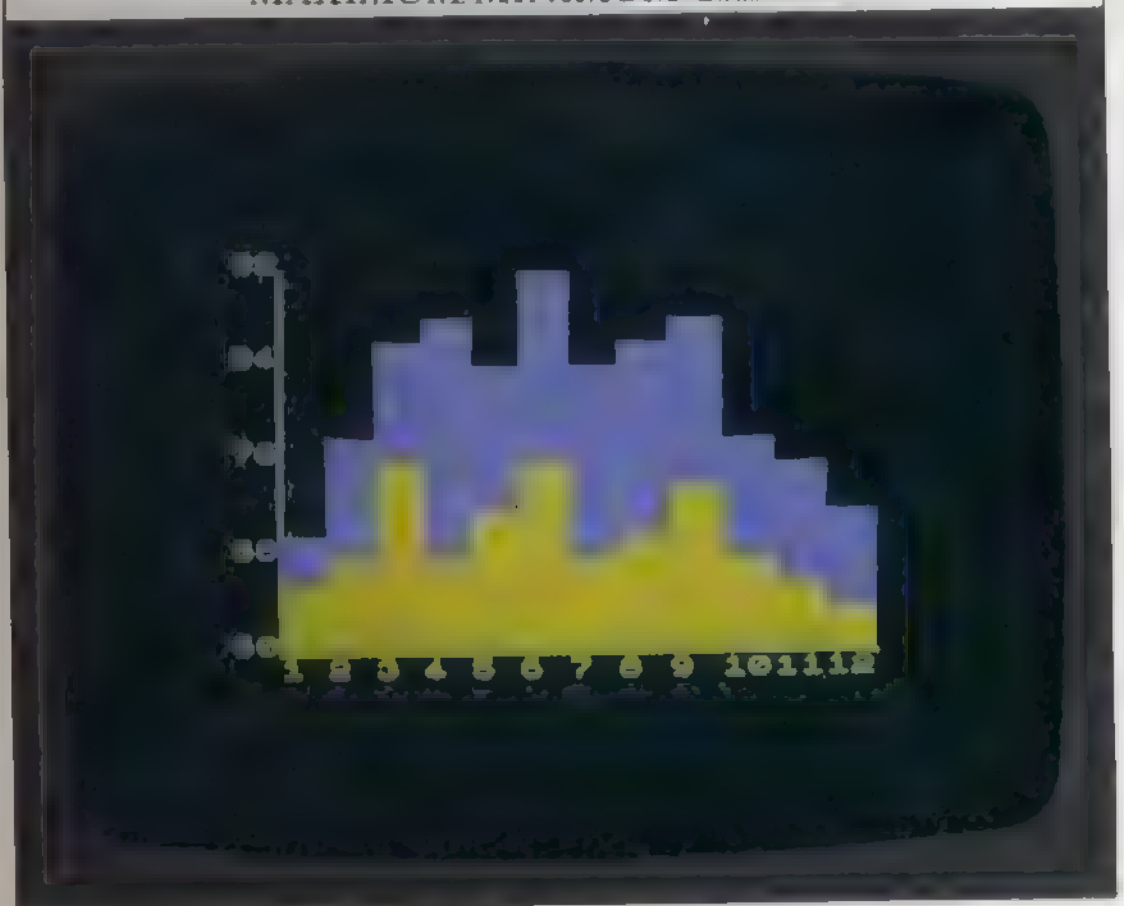
The bar charts so far have shown just one list of items. But it is possible to reorganize the first program so that it shows more than one set of information. You may for instance want to show both maximum and minimum figures like temperatures on the same chart. You don't have to rewrite the first program from scratch. A few additions will do the job:

```
105 FOR n=1 TO 2
150 IF n=1 THEN PRINT INK 3; AT r,2*m+2;
    "█"
155 IF n=2 THEN PRINT INK 6; AT r,2*m+2;
    "█"
170 NEXT n
```

This RUNs as before until you have finished keying in the first set of data. It then sets n to 2 and PRINTs

columns of yellow squares instead of magenta ones. The second set of data must be composed of figures smaller than the first set, otherwise the magenta chart will be overwritten by the yellow squares:

MAXIMUM/MINIMUM CHART



As soon as you start ENTERing the second set of items, you'll notice that the x axis and part of the y axis of the chart disappear. You can get around this very easily by reDRAWing the axes each time a column is DRAWn:

156 INK 7:PLOT 224,24:DRAW −192,0:DRAW 0,128

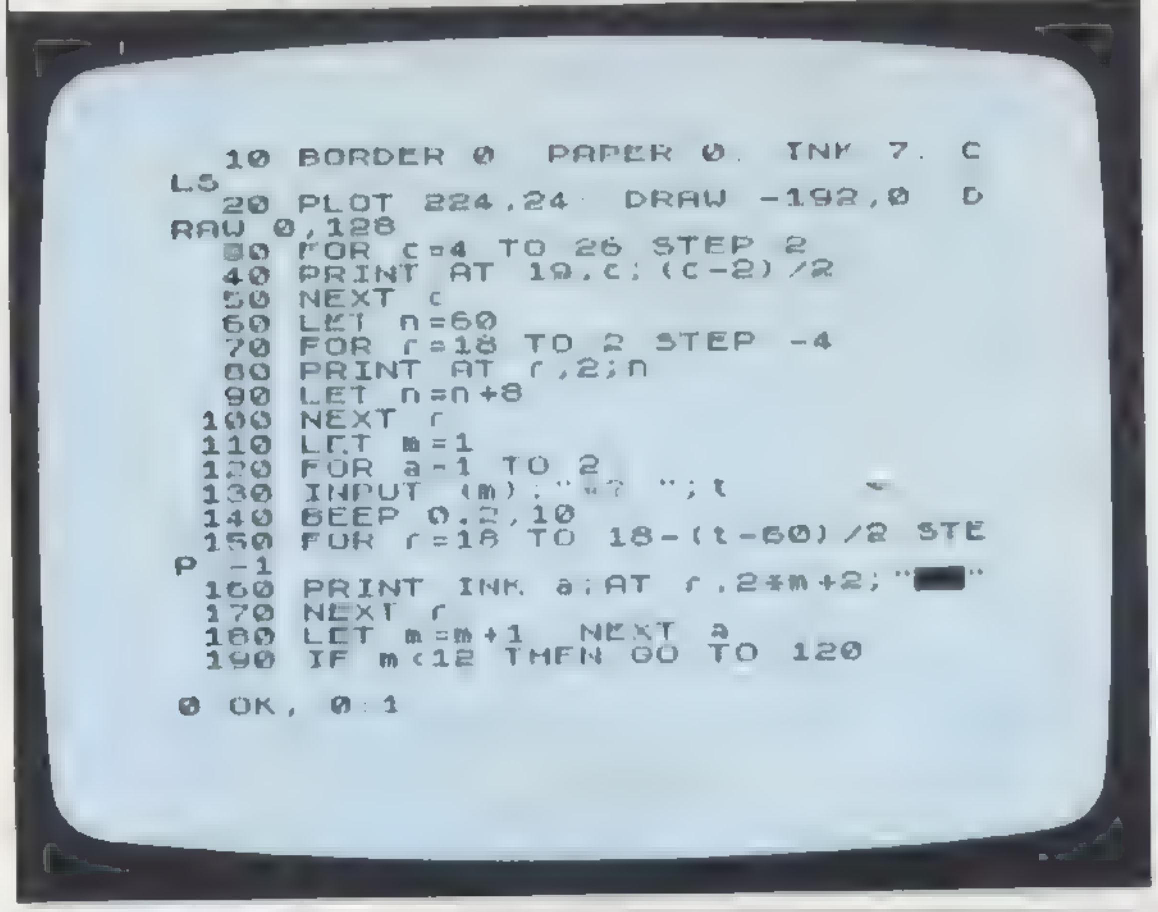
You will need to PRINT the columns as PAPER. This slows down the program considerably, so you may prefer to reDRAW the axes only once at the end, ENTERing the above line as line number 180.

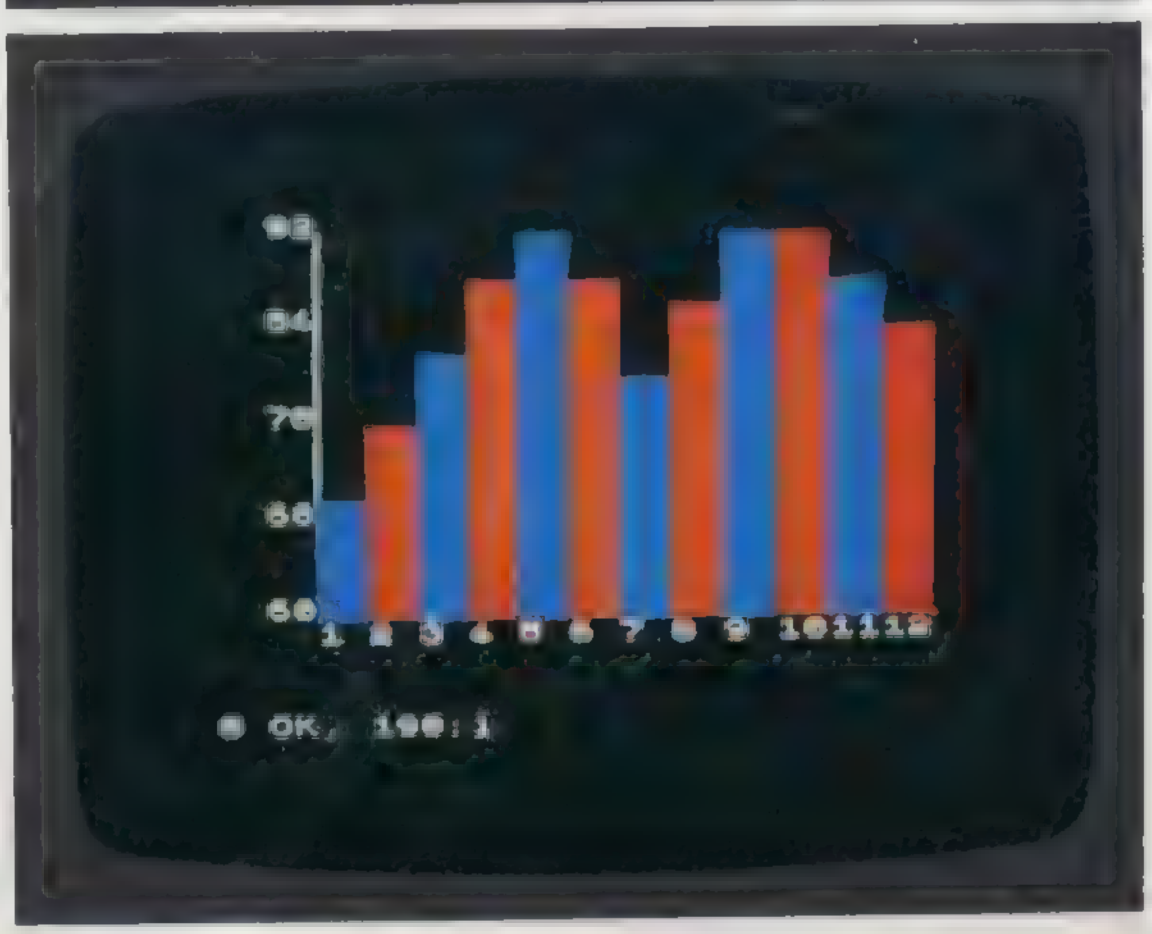
Splitting bars by changing INK

One of the problems with the previous displays is that you cannot distinguish each bar on the charts, making it difficult to connect the bars with the scale on the x axis. You can get around this by using two different INK colours again, but this time alternating them as the bars for one set of INPUTs are PRINTed. It is then quite easy to see which bar relates to which figure on the x axis.

The following program is an adaptation of the first one. If you take out the lines used for the double display, you can then edit the first program to produce the following one. Instead of having the INK colour fixed, it is now controlled by a variable a. A loop is used in conjunction with IF ... THEN to set the INK colour to either blue or red. When a is 1, the INK colour for that bar is blue, then for the next INPUT, when a becomes 2, the INK colour changes to red. You can use this type of colour-changing loop with as many of the INK colours as you like. If you want to increase the number of bars that can be made to appear on a chart, you can reduce the width of each bar by using a single graphics square only. The program would also have to be altered to change the PRINTing positions. Here is the two-colour chart program and some sample displays that it can produce:

TWO-COLOUR CHART PROGRAM



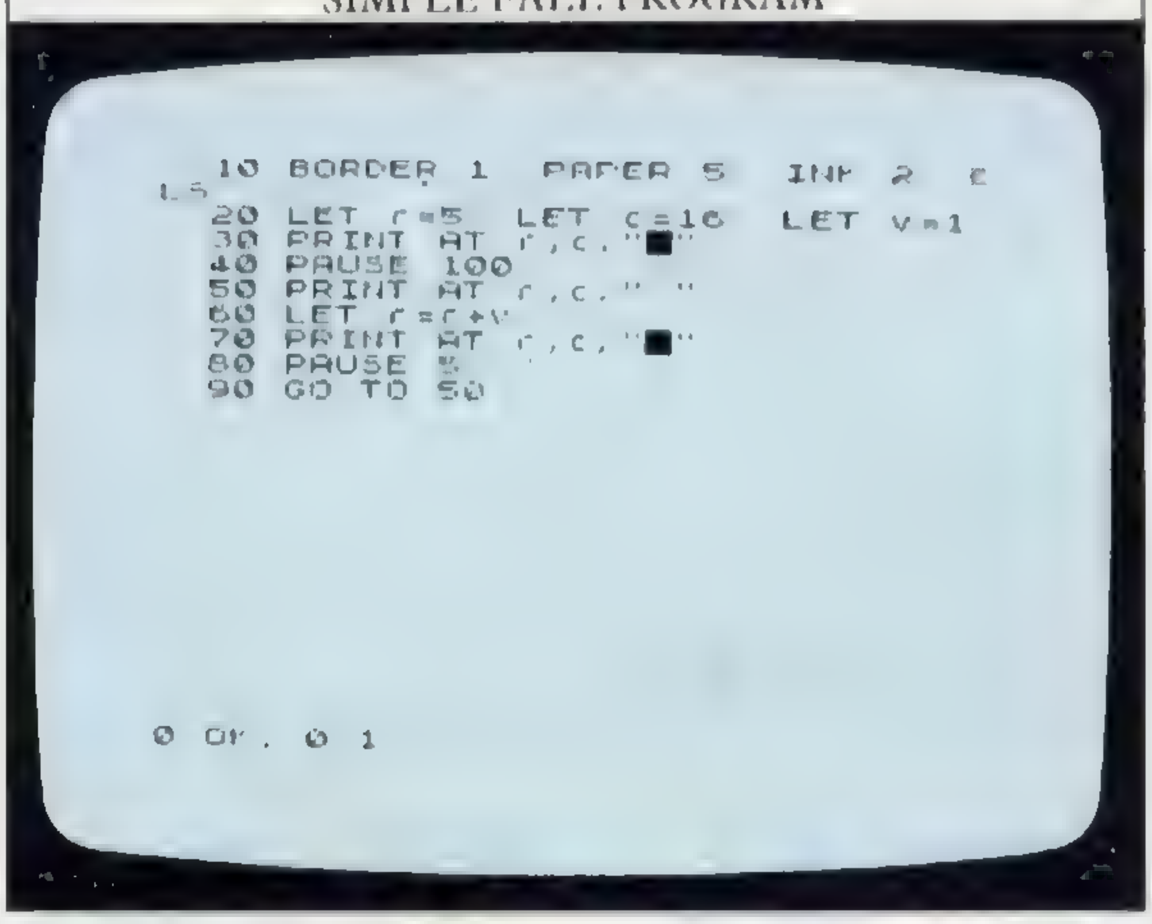


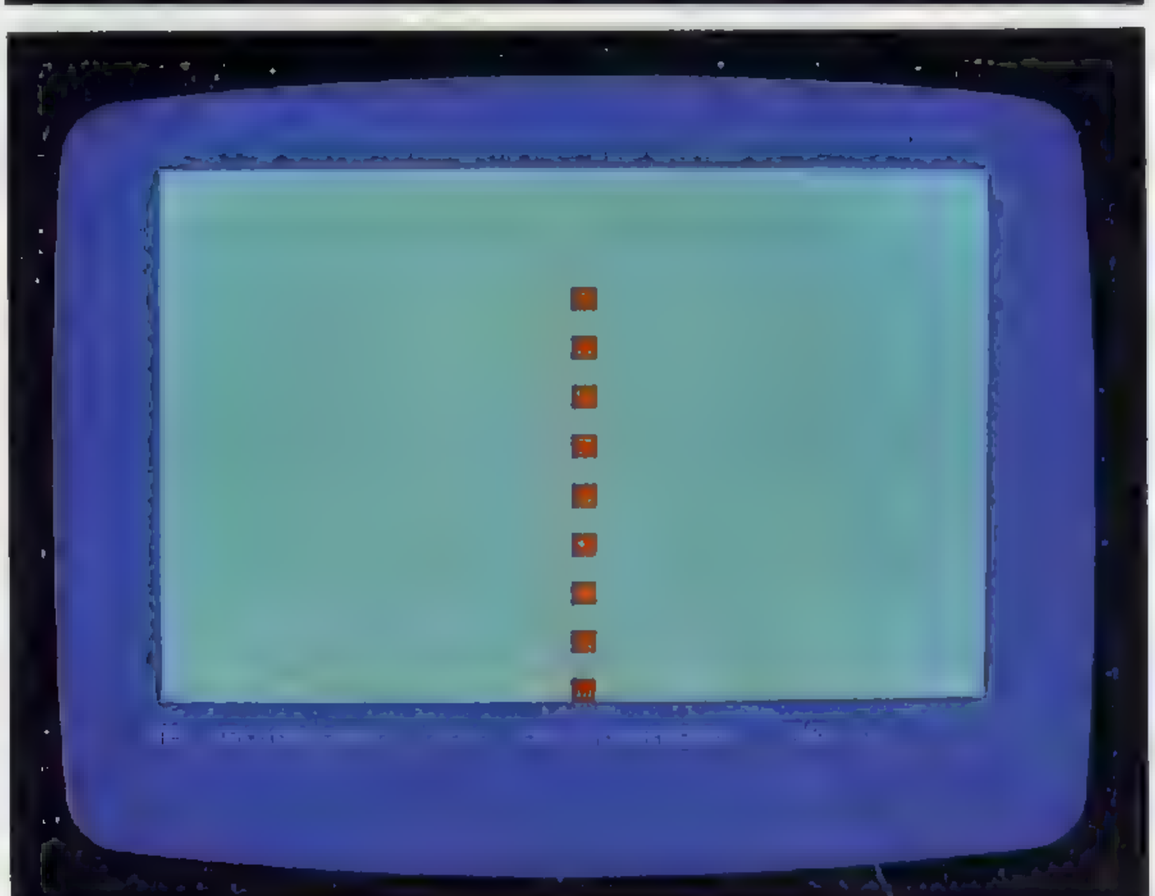
GRAPHICS WITH GRAVITY

On pages 18–19 you saw how SIN and COS could be used to make "natural" graphics, shapes that are sometimes seen in the natural world. To make these shapes you can just experiment with the graphics commands and see what happens. But if you want the computer to simulate something moving in a realistic way, an understanding of how it moves in real life will help you a great deal when you are trying to simulate that object's movements on the computer screen.

Let's take a simple simulation using a bouncing ball and go through the steps necessary to build up different types of program. On pages 8–9 you saw how IF … THEN could be used to "bounce" a ball in straight lines moving at constant speed. However a ball doesn't move in straight lines. On the screen below is a short program to demonstrate how you could begin simulating a more realistic fall (the display beneath it includes after-images normally deleted):

SIMPLE FALL PROGRAM





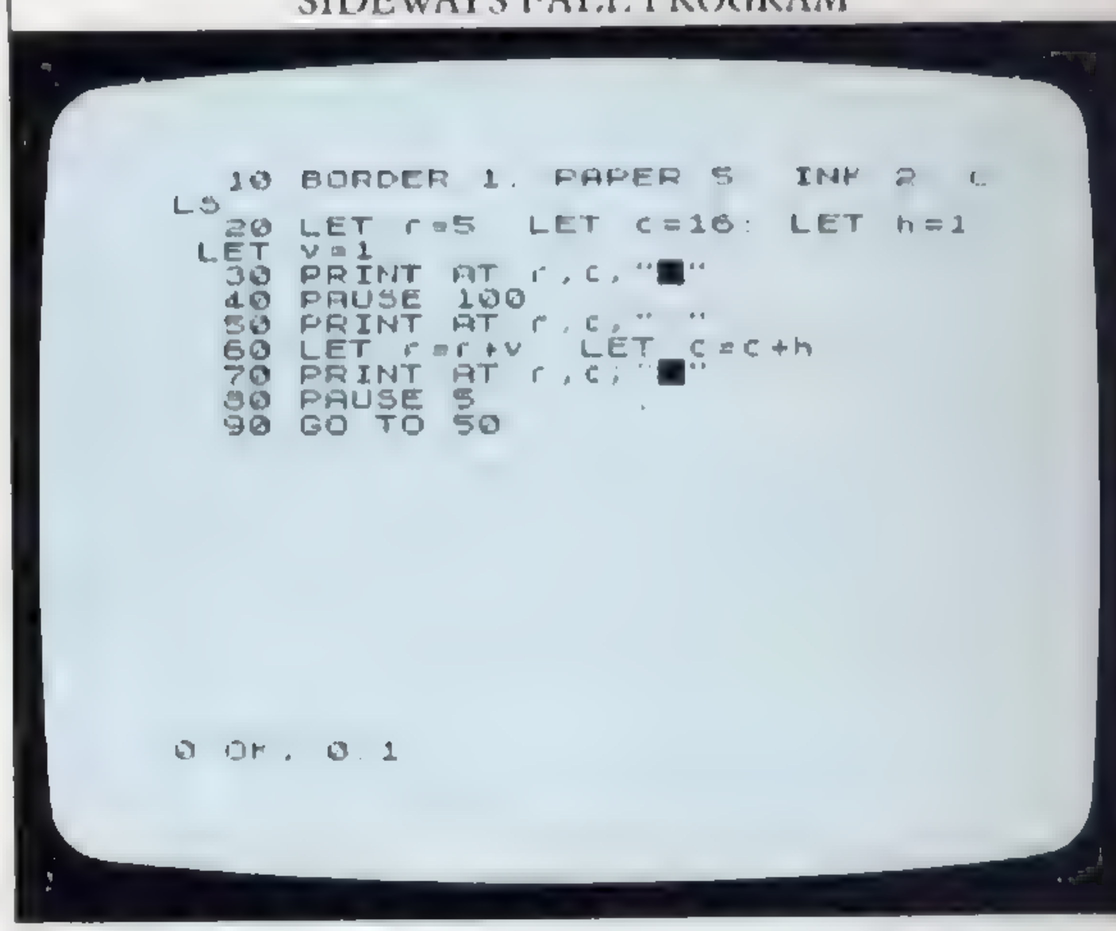
Falling objects are influenced by several forces – gravity, air resistance, surface friction and something called the "coefficient of restitution", which make them move in a complex way. However, you don't have to be a physicist to write a more realistic "bouncing" program. If you drop a ball, it falls to the ground and bounces up again, and that's all you need to know to simulate bouncing on the screen.

In the previous program listing, line 30 PRINTs the "ball" near the top of the screen. After a 2-second PAUSE, the "ball" starts to move downwards. Line 50 erases it. Next, the row number is increased by 1 and last, the "ball" is PRINTed again.

Movement in two directions

If you RUN this program, you will find that although the "ball" is indeed falling to the bottom of the screen, its movement doesn't look very realistic. The program also ends with an error message when the "ball" falls out of the bottom of the screen. The next program improves the display considerably by making the "ball" move sideways as well:

SIDEWAYS FALL PROGRAM



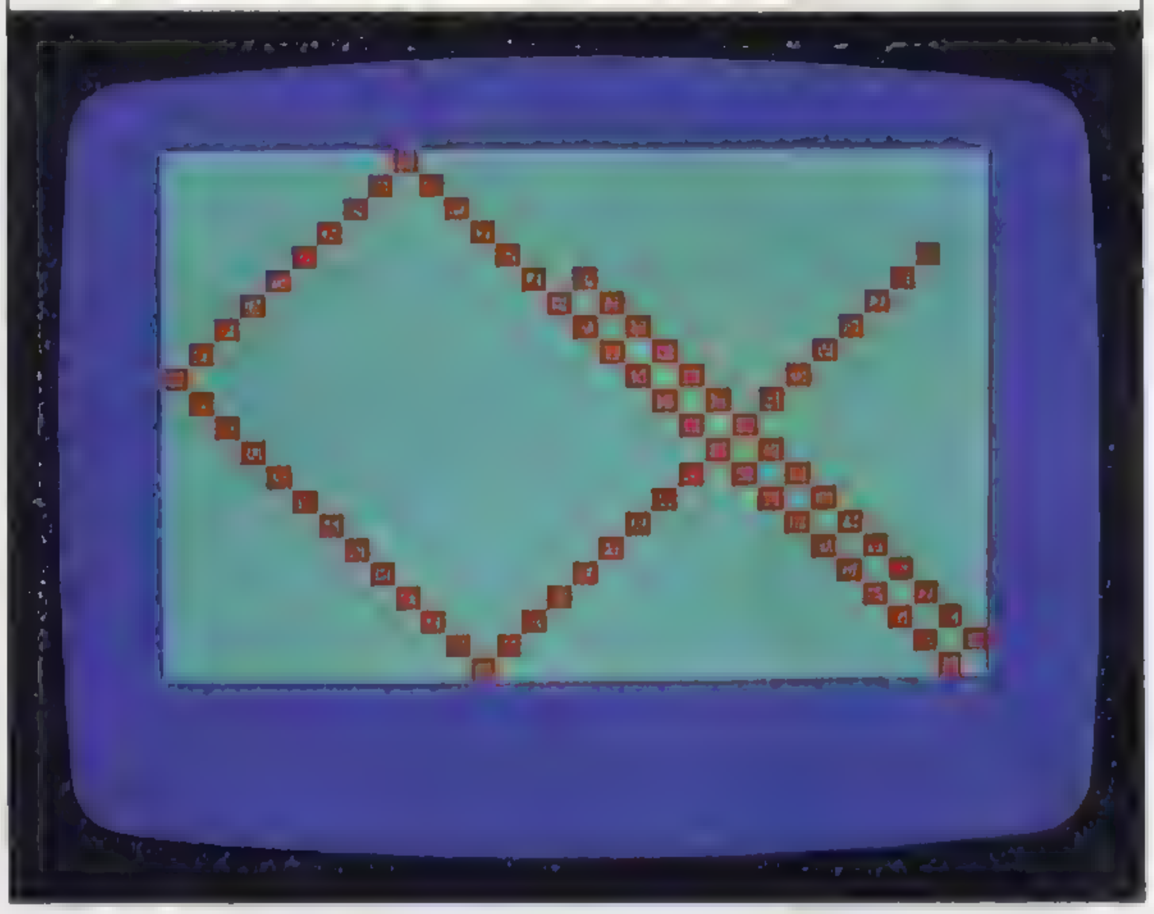
The variable h represents the change in horizontal position and v the change in vertical position. On each loop, v is added to the row number and h to the column number. Now it's easy to modify the motion in any direction. For instance, you can make the "ball" bounce by adding:

```
65 IF c=0 OR c=31 THEN LET h=-h:BEEP 0.05,20
66 IF r=0 OR r=21 THEN LET v=-v:BEEP 0.05,20
```

If you take out the lines which erase the "ball" as it moves, you will see a display like this. Note that the

BEEP is only sounded when the "ball" encounters the edge of the screen:

SIDEWAYS FALL DISPLAY

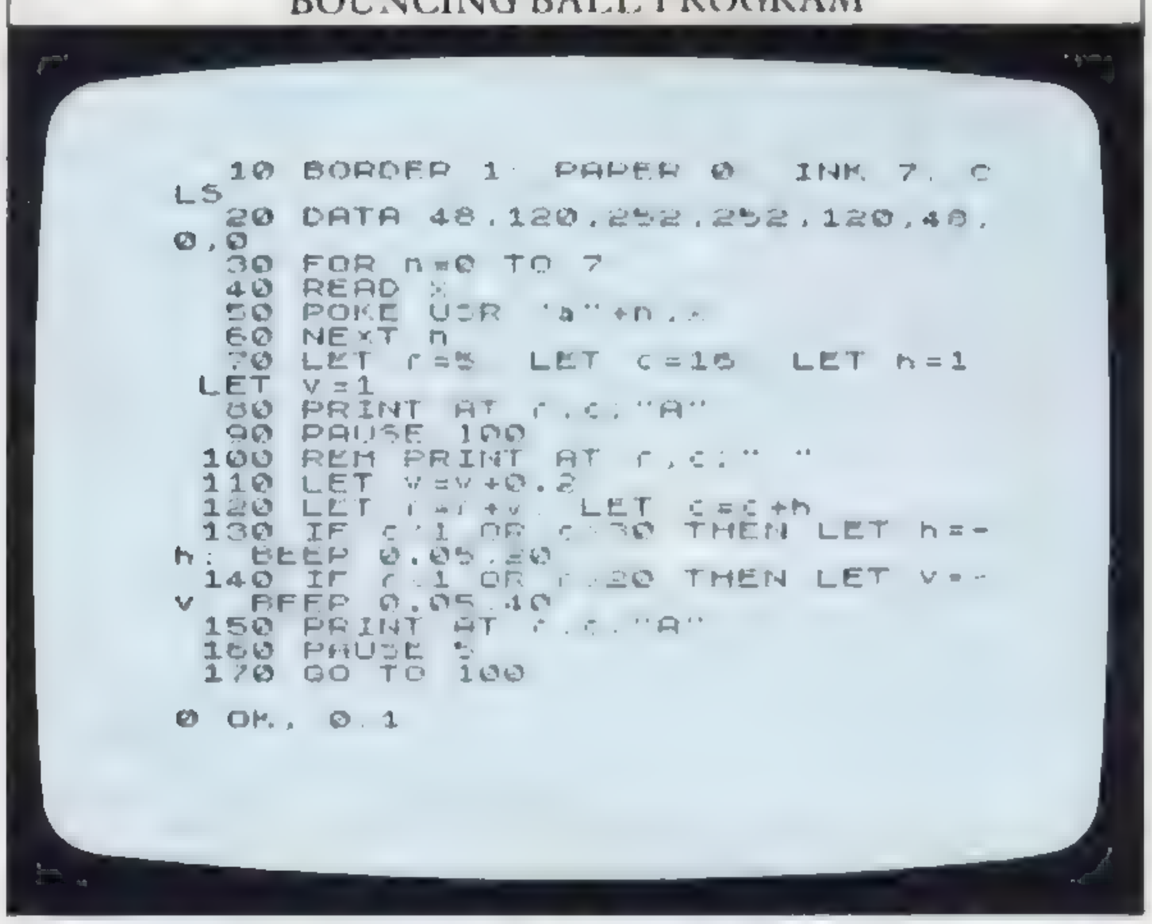


Simulating gravity

Although the "ball" now bounces around the screen, it does not yet look completely realistic. The reason for this is that the "ball" on the screen does not mimic the effects of an object falling under gravity.

You can add a "force" like gravity that acts in any direction, or that even changes direction during the program's RUN. Gravity acts downwards, so, as the "ball" moves from top to bottom of the screen it should accelerate. When it bounces up from the bottom, it should begin to slow down until it falls back down to the bottom again. The next program imitates this effect. Type in the following listing:

BOUNCING BALL PROGRAM



In this program, the ball is a user-defined character. The gravity factor is added at line 110. The addition of 0.2 to v means that the change in r – the vertical position – is no longer constant. It increases on each loop, speeding the ball up. When the ball hits the bottom of the screen its direction is reversed (line 140) and therefore v becomes a negative number, repeatedly decreasing the row number. Moreover, the additive gravity factor at line 110 makes v less and less negative, slowing down the upward progress of the ball until its vertical motion ceases, v becomes positive again and the ball begins to move downwards once again.

This display shows how the ball moves with this program:

BOUNCING BALL DISPLAY

The ball bounces around as before, but as it does so, it doesn't reach the same height on each bounce. Its height is gradually decreasing, although its horizontal movement remains the same. The result of this is a rough example of a curve known as a "parabola". Eventually the ball will bounce along the bottom line of the screen, just as a real one would.

In just the same way as the vertical movement can be modified by a "force", you can alter the horizontal movement as well. This gives the impression of an object that is not only falling under gravity, but which is also being blown along by a strong wind.

The curve that the ball makes during this program is not very smooth. This is because the ball is a text character, and its movement is limited to the 32×22 character positions on the screen. If you want to produce smoother bouncing, you can experiment with the PLOT command instead. This will produce a single point at a graphics co-ordinate, allowing much smoother curving over the 256×176 graphics grid. To do this, however, you would have to modify the program so that the character positions in all the lines were converted to graphics co-ordinates. If you refer to the grid on page 59, you should find that this is not too difficult. Because a single point is not very easy to follow, it is simplest to leave all the points PLOTted on the screen to make up a series of gravity curves, tracing the path of the point as it falls to the ground.

WRITING GAMES 1

The next six pages will take you through writing a games program, showing you how to put all the phases together to build up a complete listing. Writing a games program requires some careful planning before you actually start writing lines. To begin with, you need to decide what sort of game you want. Many games combine your acquired skill with an element of chance (the roll of dice, the turn of a card, and so on), and many have a number of different phases of play, each of which confronts you with a different set of problems.

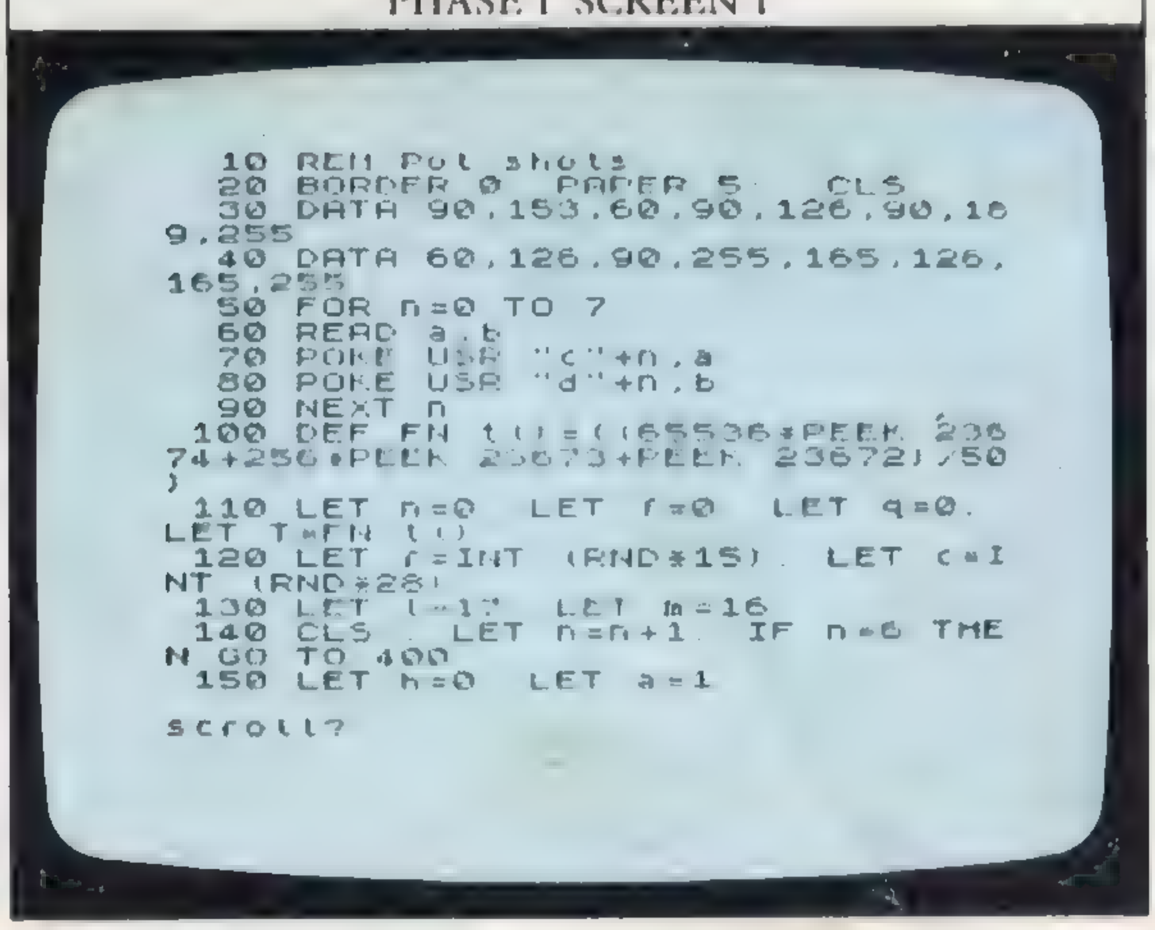
To plan a game, it is best to start by drawing a rough sketch of the screen display, marking the colours and the positions of any fixed characters or patterns. You'll want to refer back to this as you write your program.

Next, you can draw up a flowchart showing the program steps and the order in which they will appear in the program. It isn't necessary to draw a detailed chart – a list of the steps connected with arrows to show their order is sufficient. A complete games program will be more complicated than anything you've written so far, so it is worth designing the program before you key it in. It's easier to rub out a pencil arrow or a couple of lines on your plan than it is to start rearranging lines on the screen if, when you try to RUN it, you find that the program doesn't work.

Keying in phase 1

With the game on this page, the planning stage has been completed, and you can now key in the first part of the two-phase program. The listing that follows is for a practical game – one that anyone should be able to play without any prior knowledge of the program or the computer. Below is the first screen of the program. This phase of the game involves shooting at a moving spacecraft:

PHASE 1 SCREEN 1



```
  10 REM Pot shots
  20 BORDER 0: PAPER 5: CLS
  30 DATA 90,153,60,90,126,90,16
9,255
  40 DATA 60,126,90,255,165,126,
165,255
  50 FOR n=0 TO 7
  60 READ a,b
  70 POKE USR "c"+n,a
  80 POKE USR "d"+n,b
  90 NEXT n
 100 DEF FN t()=((65536*PEEK 236
74+256*PEEK 23673+PEEK 23672)/50
)
 110 LET n=0: LET f=0: LET q=0:
LET T=FN t()
 120 LET r=INT (RND*15): LET c=I
NT (RND*28)
 130 LET l=17: LET m=16
 140 CLS : LET n=n+1: IF n=6 THE
N GO TO 400
 150 LET h=0: LET a=1

scroll?
```

The program gives you a laser base which you can then move left or right. You can fire, but only straight up the screen. A number of spacecraft approach you one by one, and you must destroy them to carry on.

The DATA for the user-defined spacecraft and laser base are READ in by the loop at lines 50 to 90. The time taken to complete the game will be used later to calculate the score, so a time function is defined at line 100, using a technique mentioned on page 21. Some of the variables used in the program are initialized (set to their starting values) by line 110.

The program is impossible to decipher if you don't know what these variables represent. The following table outlines what each of them does.

PHASE 1 VARIABLES

The first phase of the game uses a total of fourteen different variables to control graphics and record strikes.

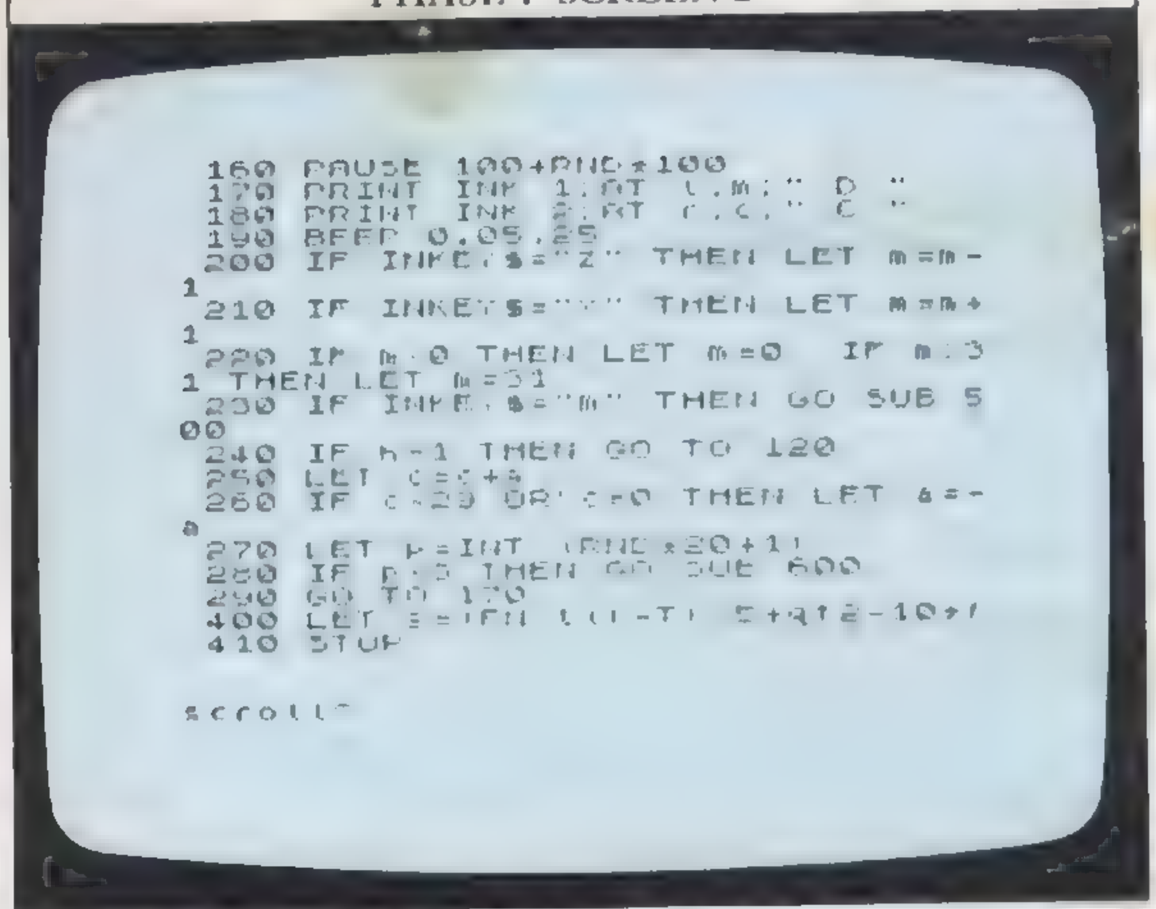
Variable(s)	Function
r,c	Fix row and column co-ordinates of spacecraft
l,m	Fix row and column co-ordinates of laser base
h	Records successful laser strike
f	Records total number of laser strikes on target
q	Records number of times laser fired
x,y	Fix starting point of laser beam
a	Sets change in position of spacecraft (=1 if left to right, otherwise = −1)
g	Sets column position of mine explosion
T	Records time taken to complete game when subtracted from FN t( )
n,p	General variables

Line 120 sets the random starting point for the spacecraft. Line 130 starts your laser base off in the middle of row 17. In line 140, n records the number of spacecraft attacks. When there have been five attacks, n will equal 6. When it does, the program jumps to line 400 and the scoring calculation.

The second screen of the program contains a number of lines which make decisions and then direct the program to later subroutines. You will notice as you go through the listing that the line numbers sometimes jump by more than 10. This is because it is simpler to give subroutines line numbers that are easily remembered – multiples of one hundred are convenient.

When you key in the second screen's lines, remember that D and C in lines 170 and 180 represent user-defined characters. This means that you will need to switch to the graphics cursor so that they are not PRINTed as letters (after the program has been RUN they can then be LISTed in the form in which they appear in the game). The second screen looks like this:

PHASE 1 SCREEN 2



After a random PAUSE (line 160), the laser base and spacecraft appear again. Lines 200 and 210 let you move your laser base to either side and line 220 stops it disappearing out of the side of the screen. If you press the M key, the program jumps to the "fire" routine at line 500. If that records a hit, the program jumps back to line 120 and begins a new attack.

Lines 250 and 260 control the movement of the spacecraft. Lines 270 and 280 make the program jump to the mining routine – which is positioned by a variable listed in the table on the opposite page – at line 600 once in every 20 spacecraft moves. This is done by picking a random number from 1 to 20; just one of these numbers will trigger the mining routine. Line 290 continues the same attack by jumping back to line 170.

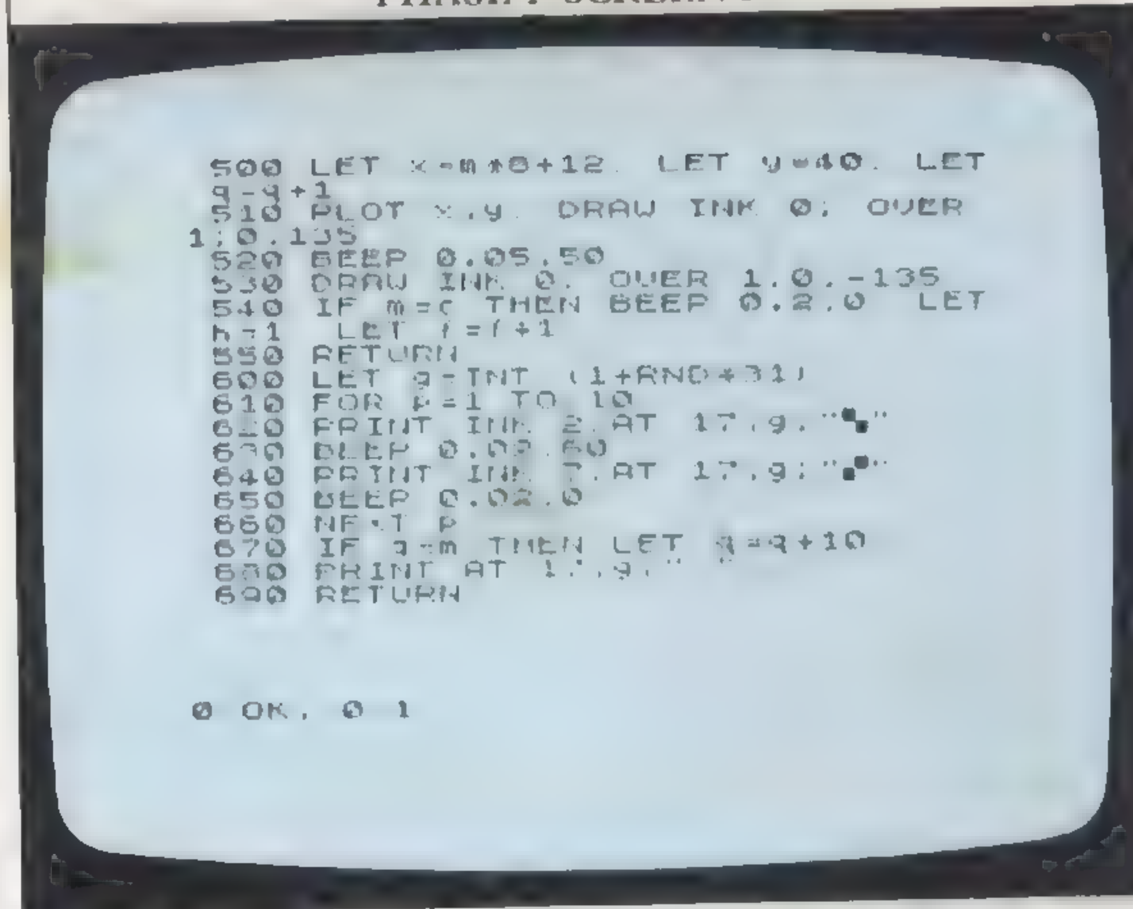
Lines 400 and 410 calculate the score and end this part of the program. The score is based on the time taken to complete the program, the number of times the laser was fired and the number of direct hits. The score isn't actually used here, but it will appear again later as you develop the game. If you want to check that the scoring lines are working, RUN the program and then key in the direct command PRINT s, without a line number. Your score should then be displayed on the screen.

The subroutine section

Finally, here is the last part of the first program. It contains a pair of subroutines. Lines 500 to 550 DRAW and "unDRAW" the laser beam using DRAW and OVER 1. If m=c then the laser has hit its target. Lines 600 to 690 explode a mine on row 17. The mine's column position is random. If it lands on the laser base, the q value of your score will be affected.

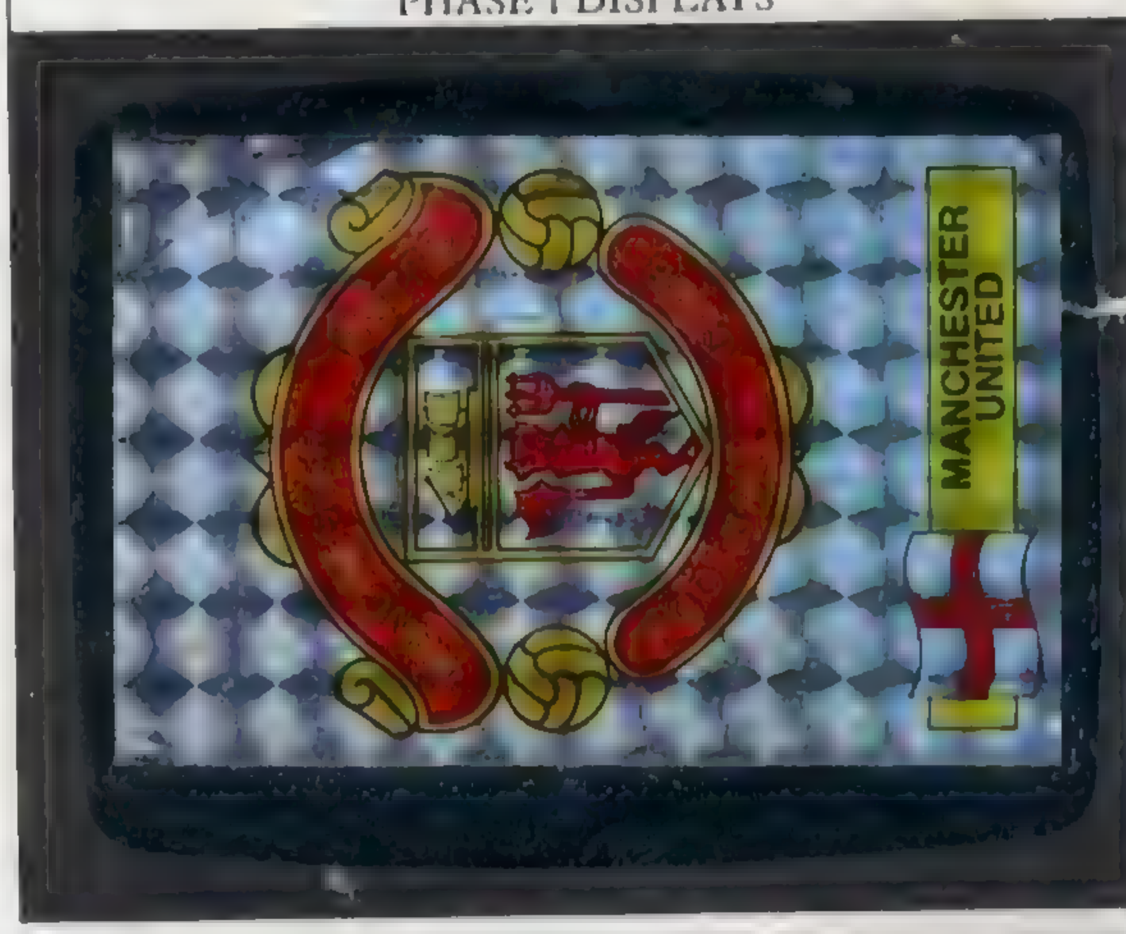
Once you have typed in the listing on the following screen and RUN this phase of the game, SAVE it on a tape so that it is ready to be combined with the next part of the program:

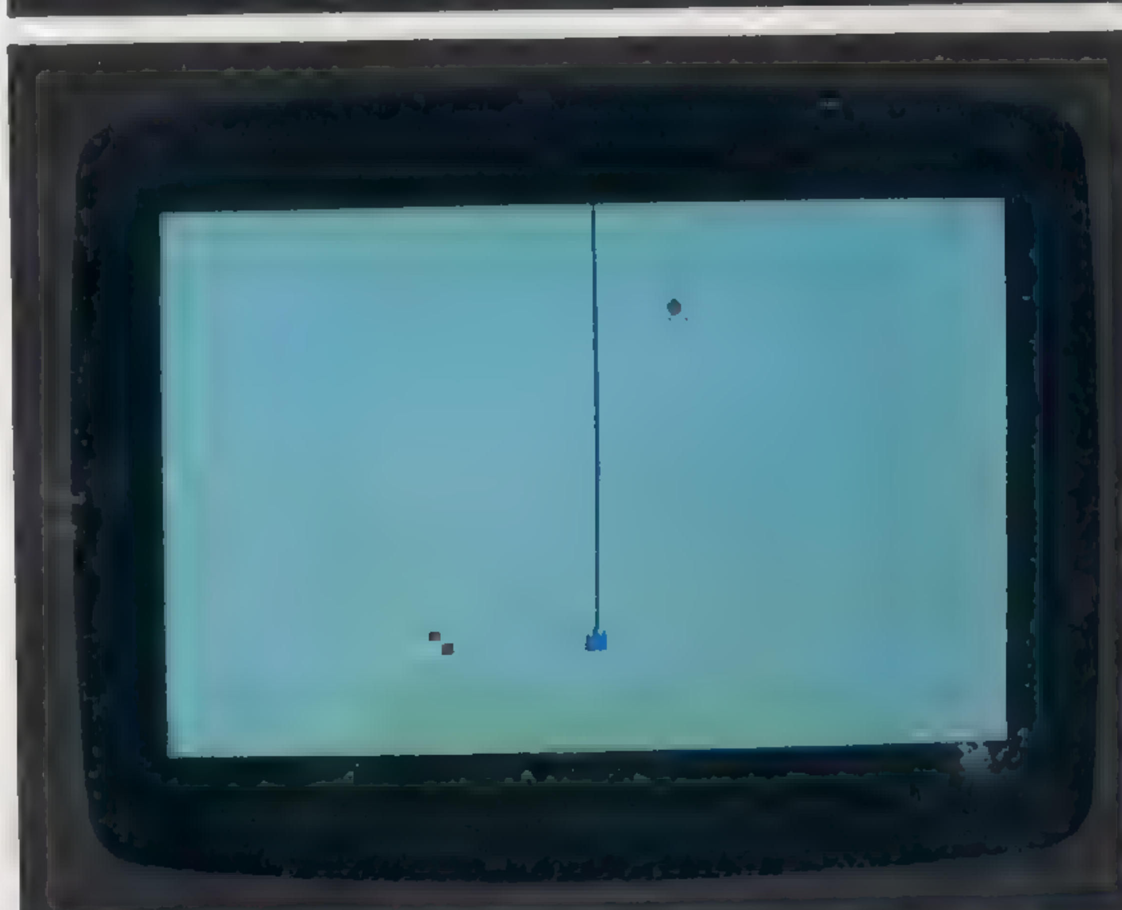
PHASE 1 SCREEN 3



Here is the program in action. In the first display, the laser is firing at the spacecraft, while in the second, a mine has appeared:

PHASE 1 DISPLAYS





WRITING GAMES 2

In the second phase of the program, the scene changes from the air to the sea as a ship tries to depth-charge a moving submarine. Again, the aim is to hit the enemy to produce the best score. The scoring instructions are still not used in this phase, but are ready to be brought into operation when the last phase of the game has been keyed in, linking up the first two parts.

As before the program uses a number of variables to control movement and subroutines. These variables need some explanation if you are to follow what is happening (in your own games you could use REM lines to remind you).

PHASE 2 VARIABLES

The second phase uses six variables to control the three objects animated by the program.

Variable(s)	Function
c	After line 1090, this fixes the column position of the ship
t,d	Fix row and column co-ordinates of submarine
f	Records when depth-charge has been dropped
u,e	Fix row and column co-ordinates of depth-charge

Setting the scene

The first section of the program produces the coloured screen, and selects some random numbers:

DISPLAY/ANIMATION SECTION

```
 100 DEF FN t()=((65536*PEEK 236
74+256*PEEK 23673+PEEK 23672)/50
): LET s=0
1010 REM Sub sinker
1020 LET n=0: LET f=0: LET T=FN
t()
1030 BORDER 2: PAPER 4: INK 1: C
LS
1040 IF n=5 THEN GO TO 1400
1050 FOR r=0 TO 4
1060 FOR c=0 TO 31
1070 PRINT AT r,c;"■"
1080 NEXT c
1090 NEXT r
1100 LET c=14
1110 LET t=10+INT (RND*11)
1120 LET d=INT (RND*11)
1130 IF INKEY$="z" THEN LET c=c-
1
1140 IF INKEY$="x" THEN LET c=c+
1
1150 IF c>25 THEN LET c=25

scroll?
```

For the moment, ignore line 100 – you will find out why it is included on the next page. Lines 1050 to 1090 PRINT a blue sky over the green sea, simply by laying down rows of blue INK squares on top of the green PAPER background. Line 1100 sets the column position for the ship and lines 1130 to 1160 control its movement across the screen.

The aim of this game is to hit the submarine. The M key controls the release of the ship's depth-charges. The ship is also manoeuverable. If you press the Z key the ship will move to the left, while pressing the X key will make it move to the right. All these functions are controlled by INKEY$ for a rapid response. The ship always starts off in the middle of the screen. The position of the enemy is less predictable. Lines 1110 and 1120 set the random starting point of the submarine. It may appear at almost any depth in the water and at any point across the screen.

Main program and subroutines

The second part of the program contains some of the main program, together with a number of subroutines which the main program calls. The subroutines control movement on the screen and detect whether or not your depth-charges have hit the target:

SUBROUTINE SECTION

```
1160 IF c<0 THEN LET c=0
1170 PRINT PAPER 1; INK 7;AT 3,c
;"  ▙  "
1180 PRINT PAPER 1; INK 7;AT 4,c
;"▀▀▀▀▀"
1190 IF f=1 THEN GO SUB 1310
1200 IF INKEY$="m" AND f=0 THEN
LET f=1: LET s=s+100: GO SUB 130
0
1210 LET d=d+1
1220 IF d>24 THEN PRINT AT t-1,d
+1;"  ";AT t,d;"       ": LET d=0
1230 PRINT AT t-1,d;"   ▌   "
1240 PRINT AT t,d;" ▀▀▀▀▀▀ "
1250 GO TO 1130
1300 LET u=5: LET e=c+2
1310 PRINT AT u,e;" "
1320 LET u=u+1
1330 IF u=21 THEN LET f=0: RETUR
N
1340 PRINT AT u,e;"▄"
```

```
1350 IF u=t AND e>d AND e<d+6 TH
EN BEEP 0.2,0: LET n=n+1: LET f=
0: GO TO 1030
1360 RETURN
1400 LET s=s+FN t()-T





0 OK, 0:1
```

Line 1200 makes the program jump to the depth-charge routine at line 1300 if you have pressed M. The score, s, is also adjusted every time a depth-charge is dropped. The score is related to the time that has passed by using FN t() in the last line of the program. Lines 1210 to 1240 control the movement and appearance of the submarine. Line 1250 continues the program by returning it to line 1130 to check the keyboard for key-presses.

Lines 1300 to 1360 make a depth-charge travel down the screen. The charge is composed of the graphics character on key 3. If the charge reaches the bottom of the screen, line 1330 returns f to its original value (zero) and RETURNs to the main program. However, if the position of the depth-charge coincides with any position occupied by the submarine (line 1350), then that attack is terminated and a new one started. You will notice when you RUN the program that only one depth-charge can be released at a time. If f has been set to 1 by line 1200, when a depth-charge is dropped, line 1190 stops you from dropping another charge until f is once again equal to zero. This will be true either when the charge reaches the bottom of the screen (line 1330) or when it hits the submarine (line 1350).

The program uses keyboard graphics to make up all the objects in this display, and as you can see from the following screens, the results show the block outlines of these characters. However, it is easy to improve the game by using user-defined characters instead. To make a ship or submarine, just PRINT a row of characters together. The greater resolution that you can then achieve will considerably improve the display, although more programming lines will be needed.

Here are some displays of the game in action. In the third screen, the path of a single depth-charge is shown by putting a REM command in line 1310, disabling the PRINT command which normally erases the charge every time it moves down one position:

PHASE 2 DISPLAY

PHASE 2 DISPLAYS

You might have noticed that there doesn't seem to be any means of erasing the old unwanted images of the ship and submarine before the new images are PRINTed. As both only move one character position to either side of their current position, they can be effectively and simply erased by including a blank square (a space) to either side of the graphics.

The scoring routine

The time is once again used to calculate the score at the end of the game. In the final version of this program, the time function will be defined in the first phase of the game, so a second definition is not necessary in this part. However, if you want to check that the program is working properly, you need to have the time function so that you can PRINT s. Line 100 is put in here so the scoring routine can be tested. The timing function will be taken out again when the final version of the game is written.

When you have checked that the program RUNs, SAVE it on cassette ready to be combined with phase 1.

WRITING GAMES 3

Now that you have keyed in and SAVEd the first two phases of the game, you are ready to add the game instructions and complete the part of the program which will produce the score. The first problem to overcome is that the two programs are SAVEd as two separate files on your cassette. To RUN as one program, they need to be combined. You cannot simply LOAD one program from cassette onto another which is already in the computer's memory. If you do this, you will find that the program in memory will simply disappear, just as if you had pressed NEW.

The Spectrum has a command to deal with this problem – MERGE. It adds together the contents of two files. To see how MERGE is used, imagine that you have SAVEd both programs from pages 38–41 on tape, but that the "sub sinker" program is already in the computer's memory. If you then type:

```
MERGE " "
```

– putting the first filename inside the quotation marks – and then press ENTER and play the tape, the first program will be LOADed into the computer. This time the "sub sinker" program will not be erased, as happens with LOAD. However, MERGE only combines programs properly if their line numbers do not overlap. The extra line (100) added to the "sub sinker" program so that you could test the scoring routine will be over-written by line 100 of the first program. This is why the "sub sinker" is numbered from line 1010 onwards. If it had been numbered from line 10, it would have been over-written by the first phase of the game wherever there were lines of the same number.

The two MERGEd phases are now a single program. As a safety precaution you could now SAVE the whole program on cassette. This is well worth the trouble if you are developing a long program, because accidental deletions can otherwise take a long time to put right.

Adding the game instructions

If you RUN the MERGEd program, you will find that although it is theoretically one program, it still behaves as two separate units. When you are writing games programs in phases like this, you will need to do a little tailoring to the final MERGEd program to make it RUN through properly.

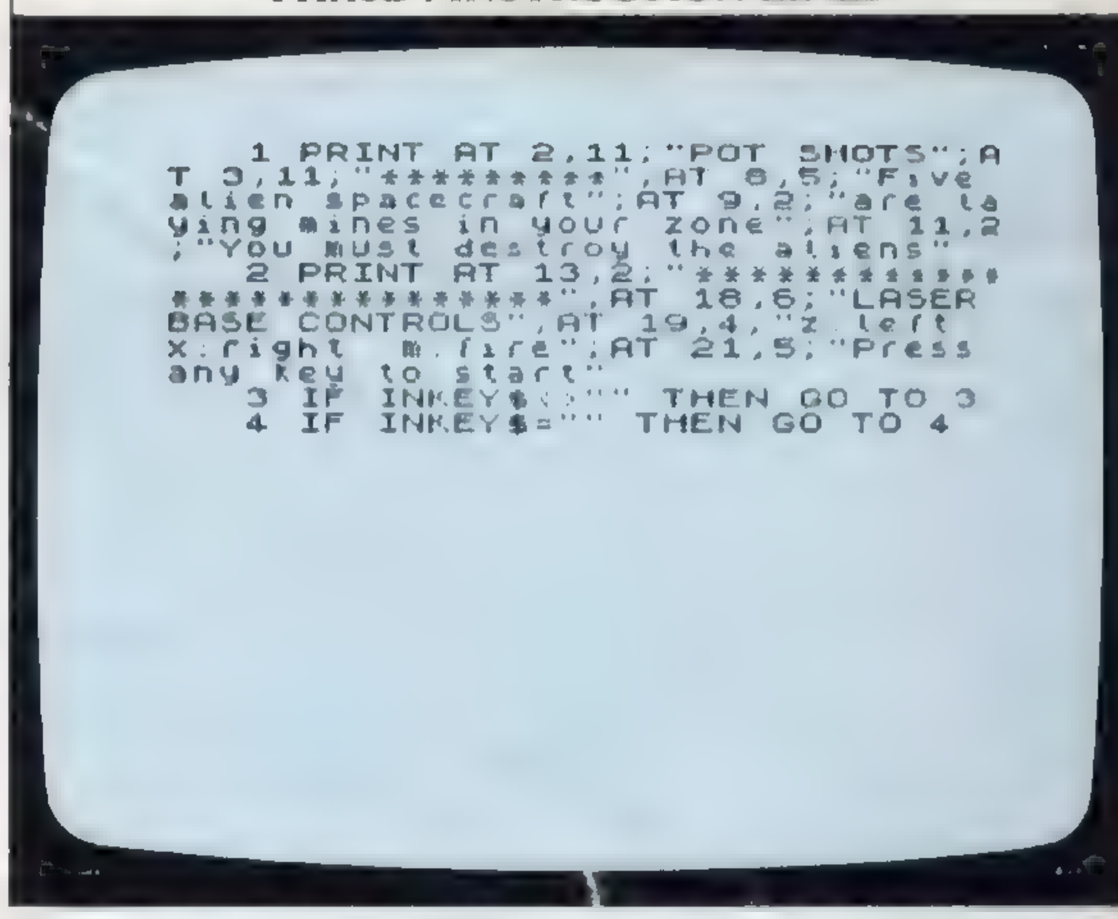
Linking the two phases is easily done. Change line 410 to:

```
410 GOTO 1000
```

That's not a mistake, even though the "sub sinker" program begins at line 1010. It's to allow you some space to add game instructions starting from line 1000.

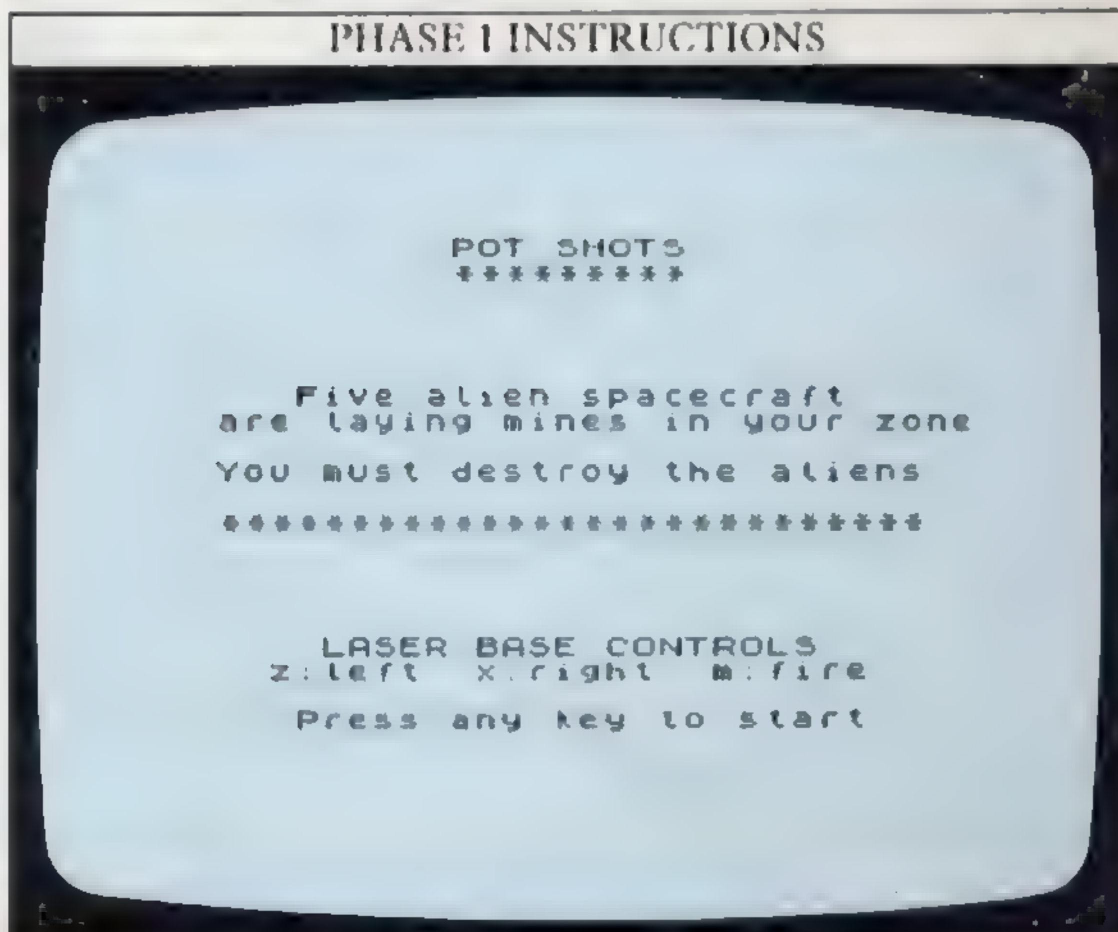
Now you can go right back to the beginning and start the program off with a title frame containing all the instructions the player will need. The keys that control the movement of objects on the screen need to be listed. You also need to tell the player how to start the game, bearing in mind that by the time the message appears, the program that contains the game will already be RUNning. Here are four lines that give the opening instructions for phase 1:

PHASE 1 INSTRUCTION LINES



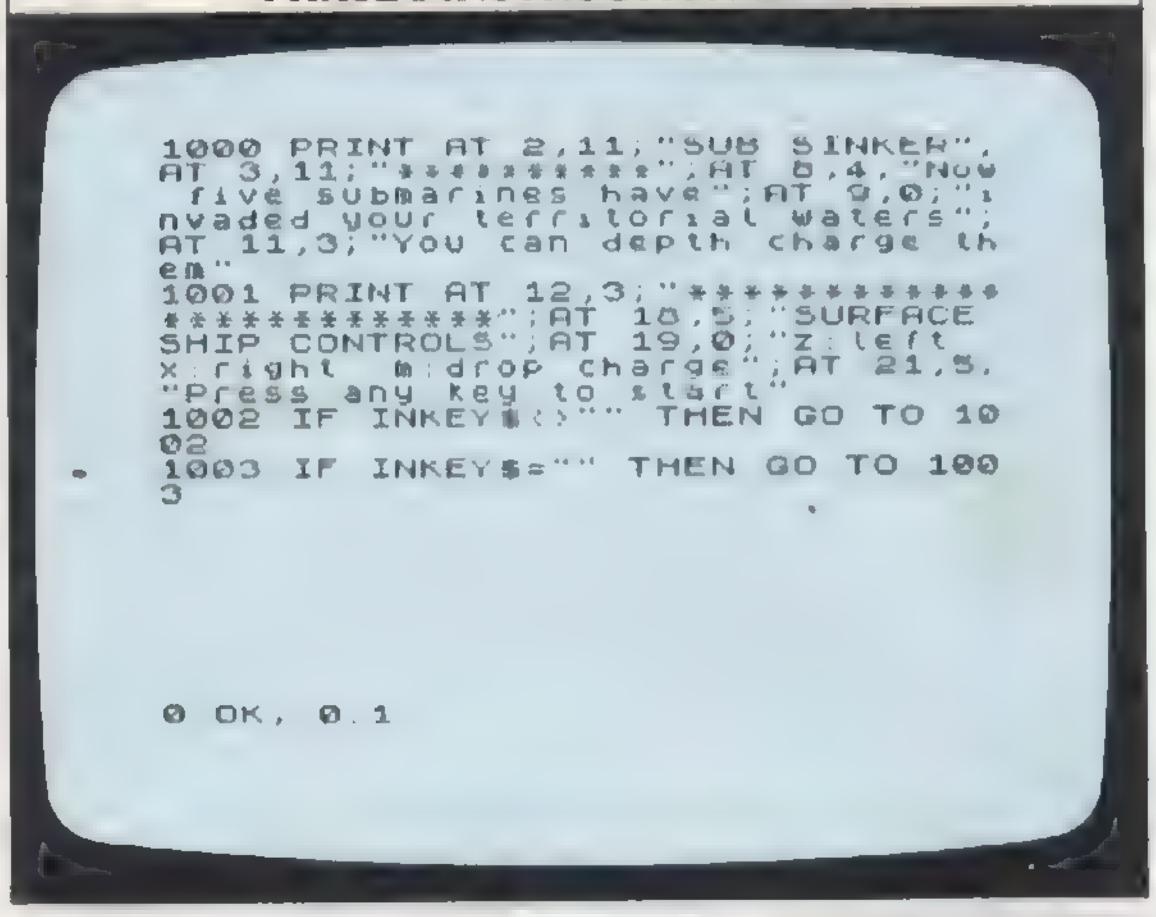
Lines 1 and 2 PRINT the game title, and explain its controls. Line 3 stops the computer from accepting your RUN and ENTER key-presses as the trigger to start the game. The next time you press a key, the condition for repeating line 4 is broken, and the first phase of the game begins:

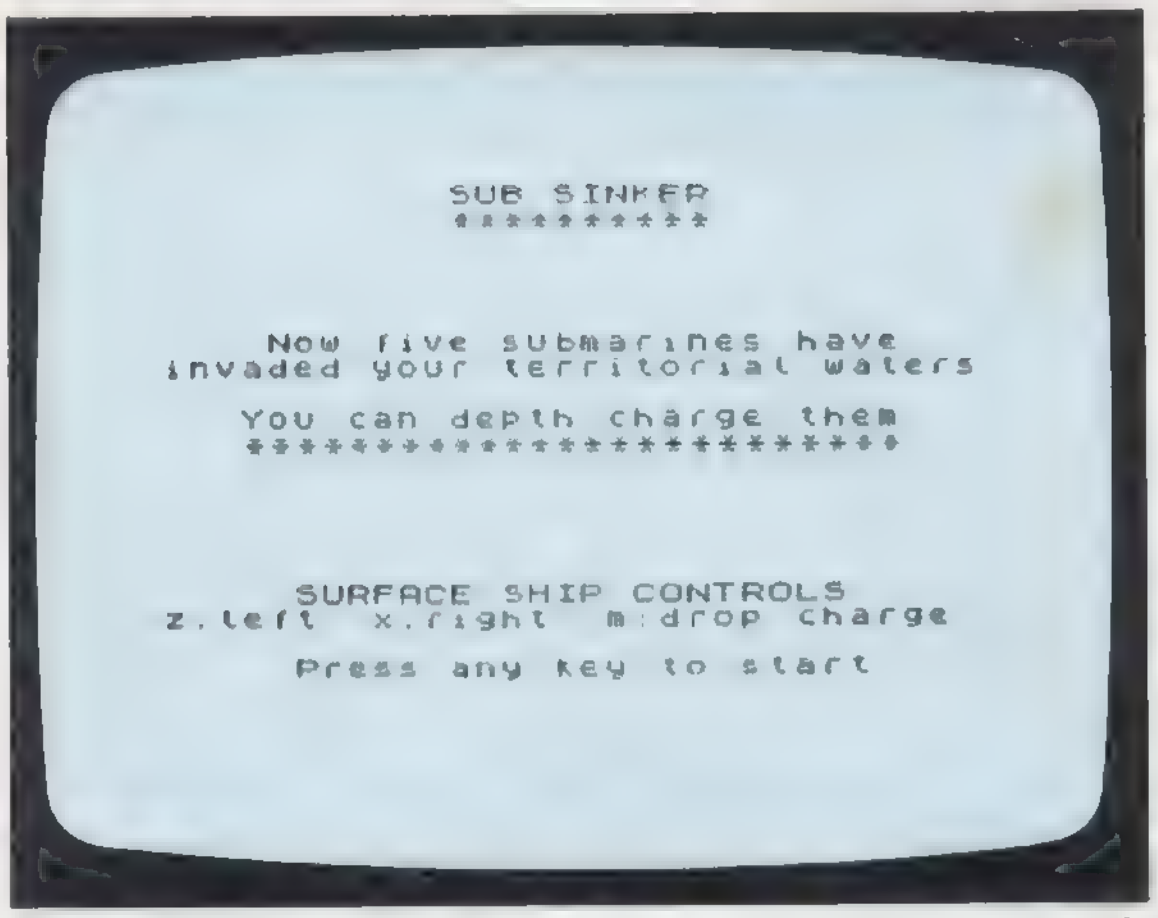
PHASE 1 INSTRUCTIONS



The instructions for the second phase of the game are inserted in a similar way:

PHASE 2 INSTRUCTION LINES





You can of course use any keys you want to specify movement as long as you change them throughout the program. Now neither phase of the game starts until the player is ready and presses a key to begin.

Completing the scoring routine

Finally, a scoring routine needs to be added to the end of the MERGEd program. The final score line of "sub sinker" is retained:

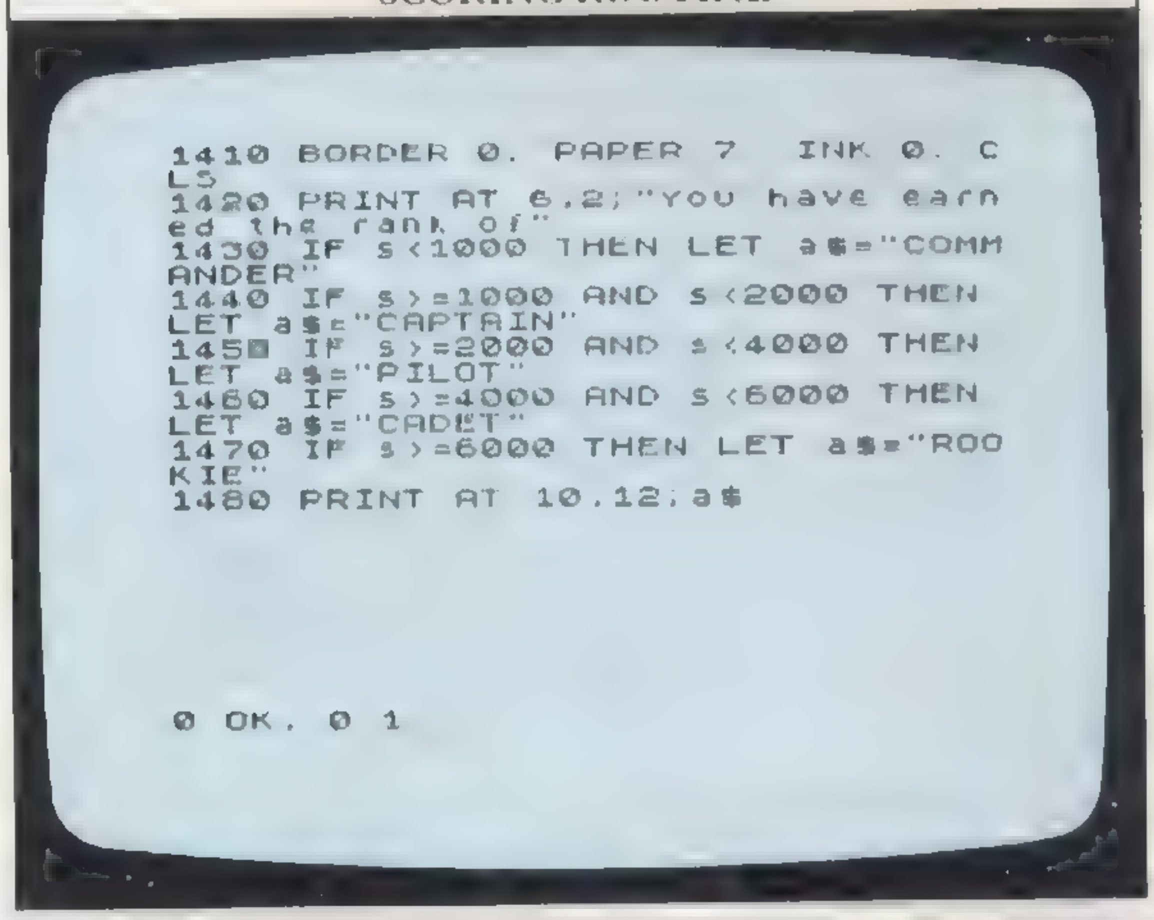
```
1400 LET s=s+FNt()-T
```

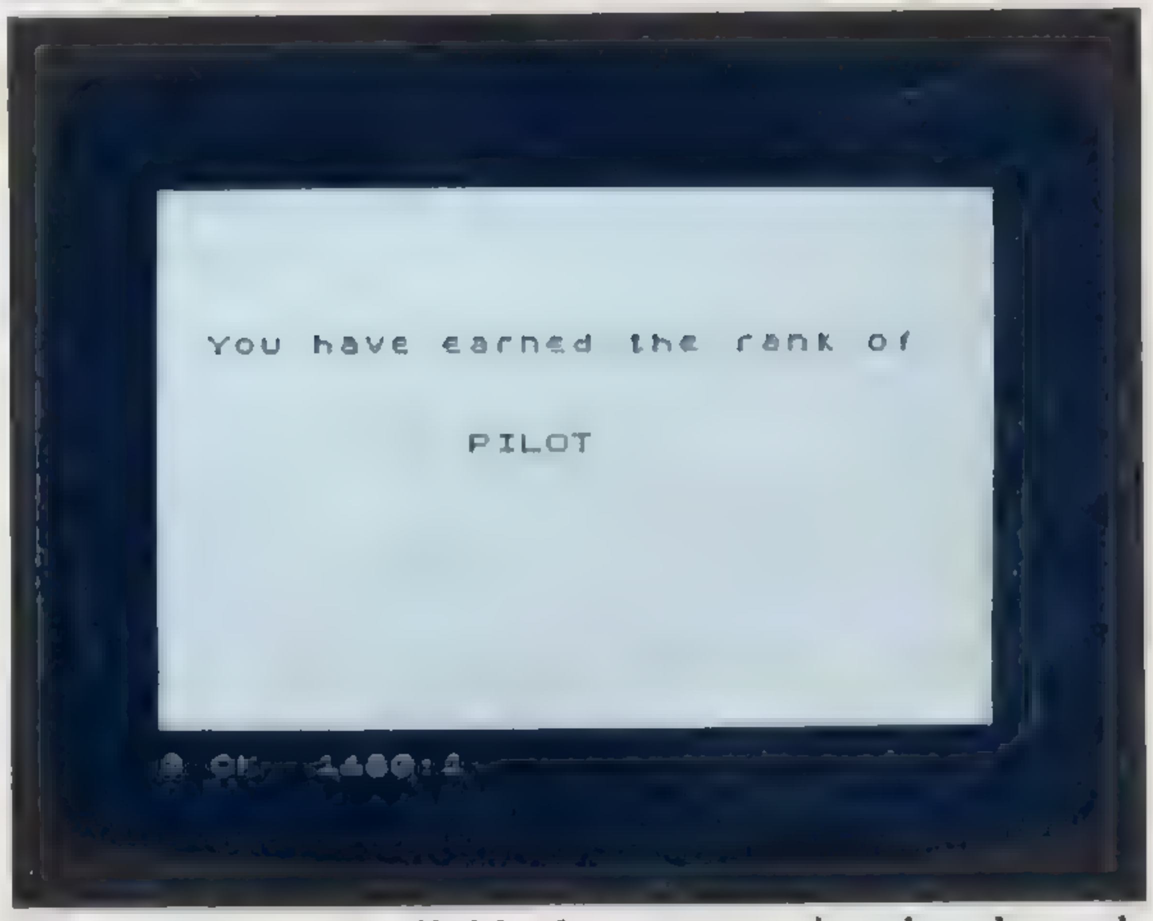
However, if you have played the two games independently and typed PRINT s afterwards, you will have noticed that the first phase of the game yields a result ranging from −20 or so to several hundred, but the second phase produces results of several hundreds to several thousands. The results of the two games need to be of the same order of magnitude. That can be achieved by multiplying the running score total in line 400 by 100. This makes the score compatible with that from phase 2:

```
400 LET s=100*(FNt()-T)/5+q↑2-(10*f)
```

This line is a good test of your understanding of the variables from pages 38–41! To make the presentation of the score more interesting, you can add a few more lines to turn this purely numerical score into a ranking:

SCORING ROUTINE





Lines 1410 to 1480 divide the scores up into bands, each of which is assigned to a rank. A series of IF … THEN lines decides where your score comes in the ranking. You can change the cut-off scores for each band to make the games harder or easier.

You now have a complete two-phase game with instructions, action and a scoring routine. Although the two phases used on these pages are relatively simple, the way that they are combined can be used to build up games of your own that are much more complex. You can use MERGE to put together a number of sub-programs, each written and tested independently. The only restriction on this is the size of the computer's memory, but unless you are combining very long programs, this shouldn't be a problem.

IMPROVING SOUND

On the Spectrum the command used to produce sound, BEEP, is quite straightforward. A line like:

100 BEEP d,p

will produce a sound that is d seconds long at a pitch p. But one of the problems of using BEEP in programs is that it stops everything else while it is being carried out. This means that if you want to produce long sounds in conjunction with movement on the screen, you cannot use a single BEEP command, or the program will "freeze" until the BEEP has finished. To link sound and animation as well as possible, BEEP needs to be used briefly but frequently, so that the sounds produced add up to give the effect you want.

How to improve sound with animation

You can hear BEEP working badly with animation if you RUN the following program:

BASIC SOUND AND ANIMATION PROGRAM

```
  10 DATA 0,130,0,3,239,128,0,68
,0,15,69,224
  20 DATA 0,40,0,30,40,240,8,254
,32,63,125,248
  30 DATA 9,57,32,63,255,248,5,2
55,64,68,0,68
  40 DATA 3,131,128,200,0,38,2,2
54,128,144,0,18
  50 FOR n=0 TO 7
  60 READ u,v,w,x,y,z
  70 POKE USR "a"+n,u
  80 POKE USR "b"+n,v
  90 POKE USR "c"+n,w
 100 POKE USR "d"+n,x
 110 POKE USR "e"+n,y
 120 POKE USR "f"+n,z
 130 NEXT n
 140 LET r=10: LET c=14
 150 BORDER 2: PAPER 3: INK 6: C
LS
 160 PRINT AT r,c;"     "
 170 PRINT AT r+1,c;" ABC "

scroll?
```

```
 180 PRINT AT r+2,c;" DEF "
 190 PRINT AT r+3,c;"     "
 200 LET a=INT (RND*2)
 210 IF a=0 THEN LET a=-1
 220 LET b=INT (RND*2)
 230 IF b=0 THEN LET r=r+a
 240 IF b=1 THEN LET c=c+a
 250 IF c<0 OR c>27 OR r<0 OR r>
18 THEN GO TO 150
 260 BEEP 0.2,25
 270 GO TO 160

0 OK, 0:1
```

BASIC SOUND AND ANIMATION PROGRAM

The flying saucer is made up from six user-defined characters arranged in two rows – a,b,c on top and d,e,f underneath, with one space either side. In addition, a row of spaces is PRINTed above and below the saucer, so that, whichever way it moves, the spaces surrounding it erase the previous image. Line 200 produces a 0 or 1 at random. Line 210 turns that into either −1 or +1. This will be used to change the position of the saucer. Line 220 produces another 0 or 1 at random. If it's a 0, the saucer's row is changed by the variable a. If the result of line 220 is 1, the saucer's column is changed by a. This moves the saucer around on the screen in an unpredictable way. Line 250 tests to see if the saucer has reached the edge of the screen and if so brings it back to the centre. The sound is dealt with in line 260 in a single BEEP statement lasting a fifth of a second. It can be improved by adding these lines:

NEW LINES TO SPLIT SOUND

```
195 BEEP 0.005,25
205 BEEP 0.005,30
215 BEEP 0.005,35
225 BEEP 0.005,40
235 BEEP 0.005,45
245 BEEP 0.005,40
260 BEEP 0.005,35

0 OK, 0:1
```

The first version is a very poor example of animation. Effective animation relies on changing from one character display to the next as quickly as possible, but in that program the single BEEP interrupts the movement for too long, so the saucer moves in a slow, halting way and the sound effect really doesn't add much to it.

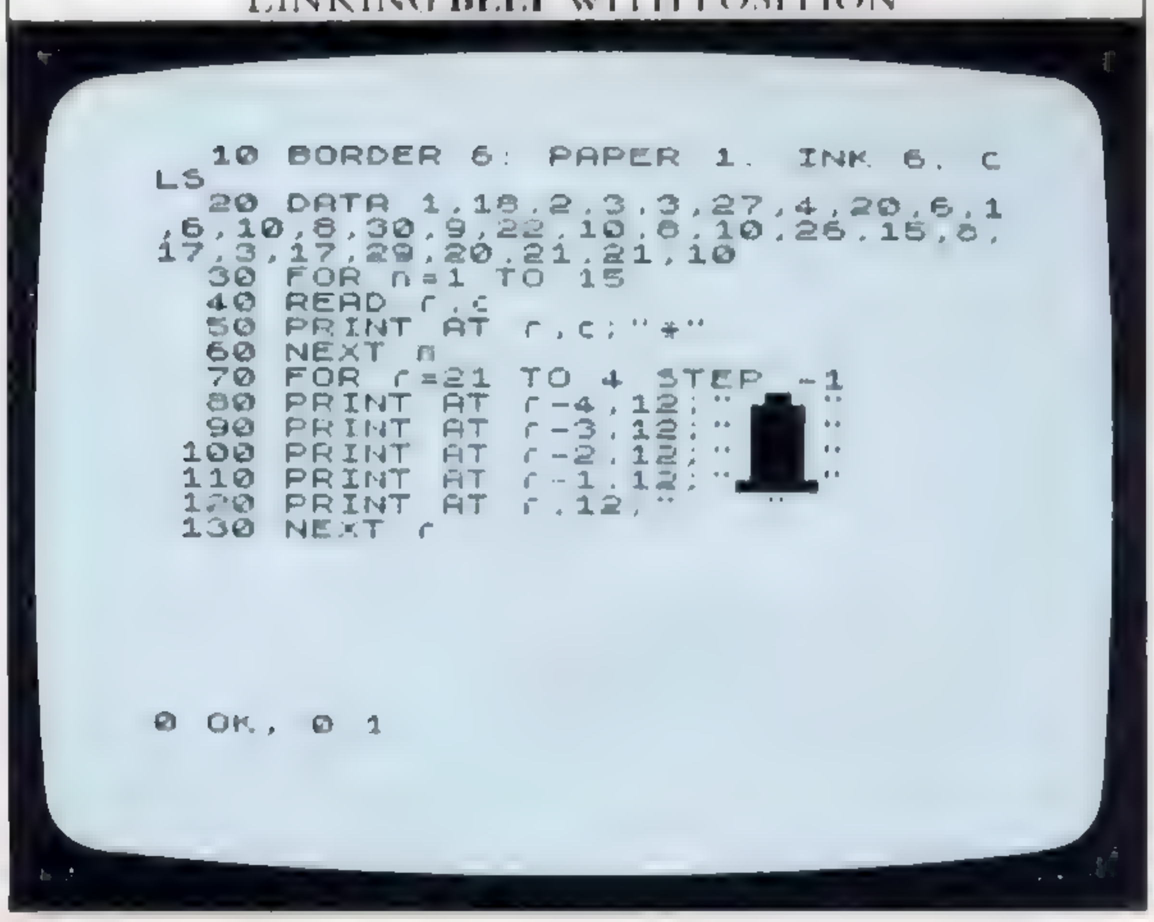
By adding the lines listed on the previous screen you will split the BEEP up into a number of shorter sounds, thus improving the quality of sound in the program. Each BEEP now lasts for only five hundredths of a second. The total duration of all the BEEP statements (0.035 seconds) is less than a quarter of the length of the single BEEP used previously, so this program RUNs approximately four times as quickly as the first version. Moreover, because the sound effect is split up into a number of separate statements, this gives you a chance to sound them at different pitches to produce a much more interesting warbling effect.

You could also write more BEEPs between the introductory lines (10 to 140) to announce the arrival of the saucer in advance of its appearance, but note that any BEEPs written into lines 50 to 130 will sound eight times because of the n loop connecting them.

Linking BEEP to screen positions

One very effective way of producing sound with animation is to link the BEEP pitch to a variable that controls a character's position on the screen. You will have already encountered this with the "bouncing" program on pages 8–9. Linking BEEP like this is fairly easy, and it cuts out quite a lot of programming lines. All you have to ensure is that the variable controlling the BEEP comes within the right range, and is used to change the duration or pitch of the BEEPs in the right direction. You can experiment with these techniques on the next program. Type in the following listing and RUN the program:

LINKING **BEEP** WITH POSITION



```
  10 BORDER 6: PAPER 1. INK 6. C
LS
  20 DATA 1,18,2,3,3,27,4,20,6,1
,6,10,8,30,9,22,10,8,10,26,15,8,
17,3,17,29,20,21,21,10
  30 FOR n=1 TO 15
  40 READ r,c
  50 PRINT AT r,c;"*"
  60 NEXT n
  70 FOR r=21 TO 4 STEP -1
  80 PRINT AT r-4,12;"  "
  90 PRINT AT r-3,12;"  "
 100 PRINT AT r-2,12;"  "
 110 PRINT AT r-1,12;"  "
 120 PRINT AT r,12;"  "
 130 NEXT r

0 OK, 0 1
```

This program PRINTs a star field and then moves a keyboard graphics rocket upwards through it. Lines 10 to 60 PRINT the star field. Lines 70 to 130 PRINT and move the rocket. When you RUN the program, the following display should appear:

LINKED **BEEP** DISPLAY

Now you can add the sound effects to the above program. This time, their aim is to produce an unusual sound of changing pitch. What is needed is a BEEP statement containing a variable whose value changes every time the BEEP is carried out. The obvious thing to relate the pitch to is the changing row number. The problem is that the sound should increase in pitch as the rocket rises, but as it does so, the row number decreases. So if the pitch were simply made a multiple of or a fraction of the row number, it, too, would decrease as the rocket rose up through the star field.

However, there is a way of overcoming this problem. Try typing in this line:

```
125 BEEP 0.05,80-(10*r)/2
```

Now the smaller r is, the higher is the pitch number. If the range of pitch is too large, multiplying r by a smaller number will reduce it:

```
125 BEEP 0.05,80-(5*r)/2
```

If the pitches are too high overall, reducing the starting value (80) will bring all the BEEPs down in pitch:

```
125 BEEP 0.05,60-(5*r)/2
```

To make the sound more interesting, you can split it into two components:

```
 75 BEEP 0.025,60-(5*r)/2
125 BEEP 0.025,63-(5*r)/2
```

The second sound is slightly higher in pitch than the first, giving a progression in pitch between the two. Using a number of even shorter BEEPs in these lines will make the sound effect even more complex.

THE SPECTRUM SCREEN POINTER

In many Spectrum games programs, areas of INK colour on the screen represent objects that you have to avoid. If you hit them, the programs respond with a penalty of some kind. But how do you go about programming the computer to decide if a character you are moving on the screen has hit something? You could use IF ... THEN to check co-ordinates, but there is a way that is much easier and quicker.

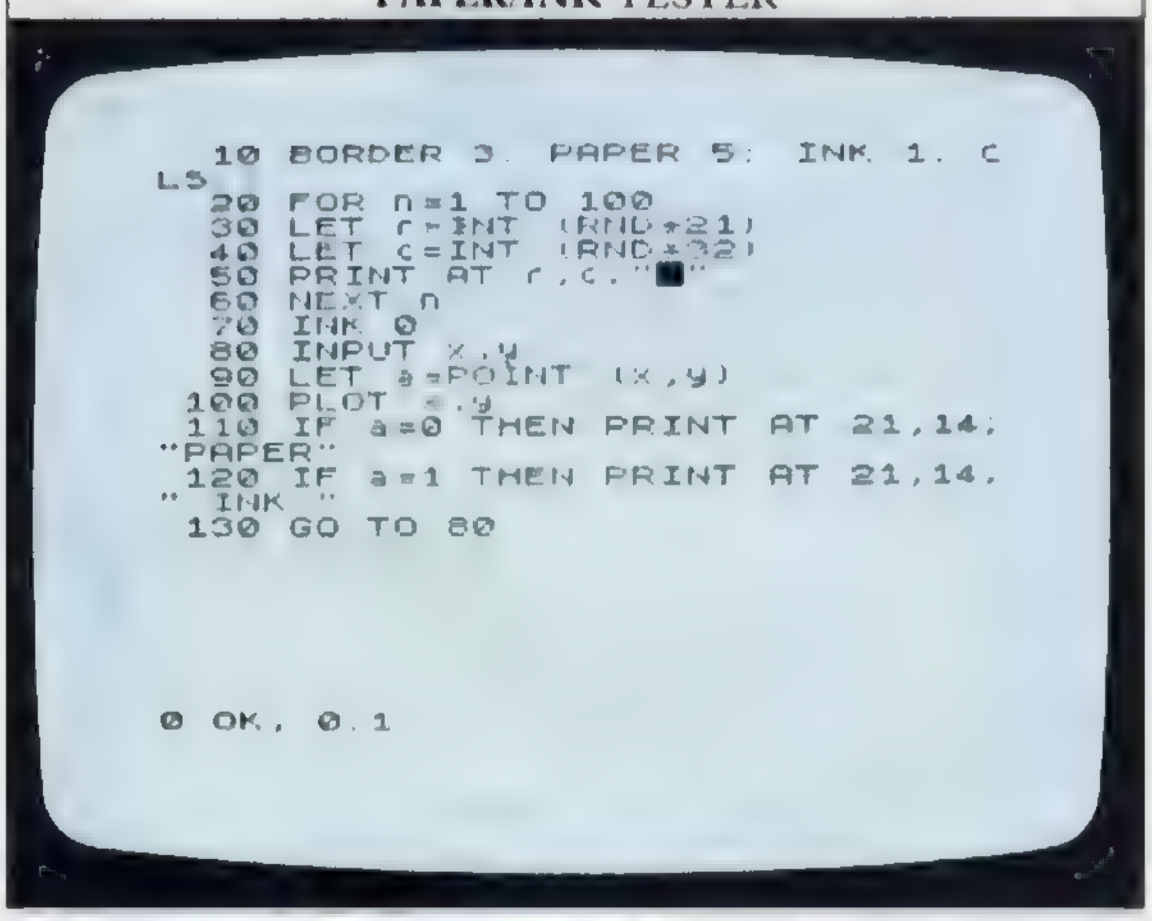
The Spectrum BASIC command POINT lets the computer examine any point on the screen and test whether it is a PAPER or an INK colour. In the line:

```
100 a=POINT (x,y)
```

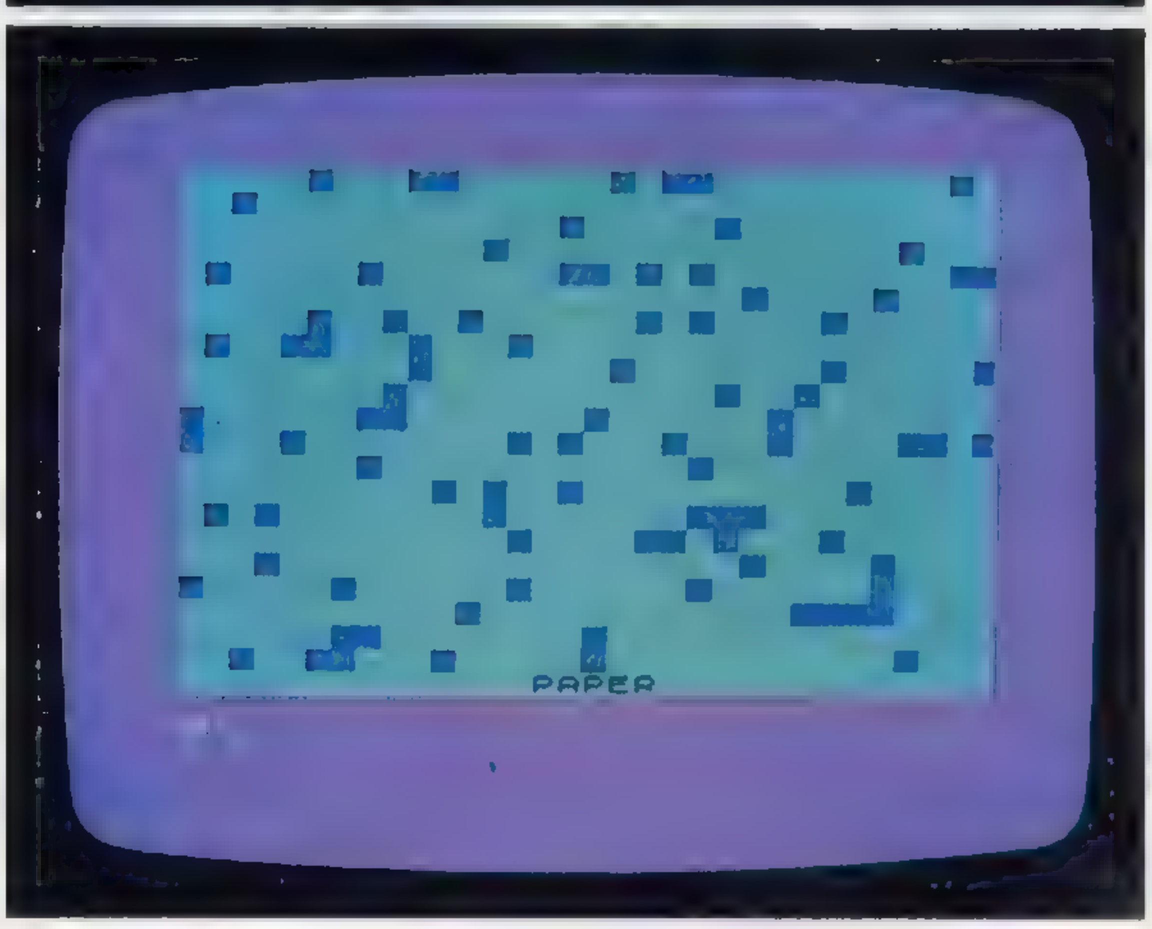
a will be equal to zero if x,y is a PAPER colour, or it will be equal to 1 if x,y is an INK colour.

The next program produces INK squares at random, and then lets you test whether a particular pair of co-ordinates is covered by a PAPER or INK colour:

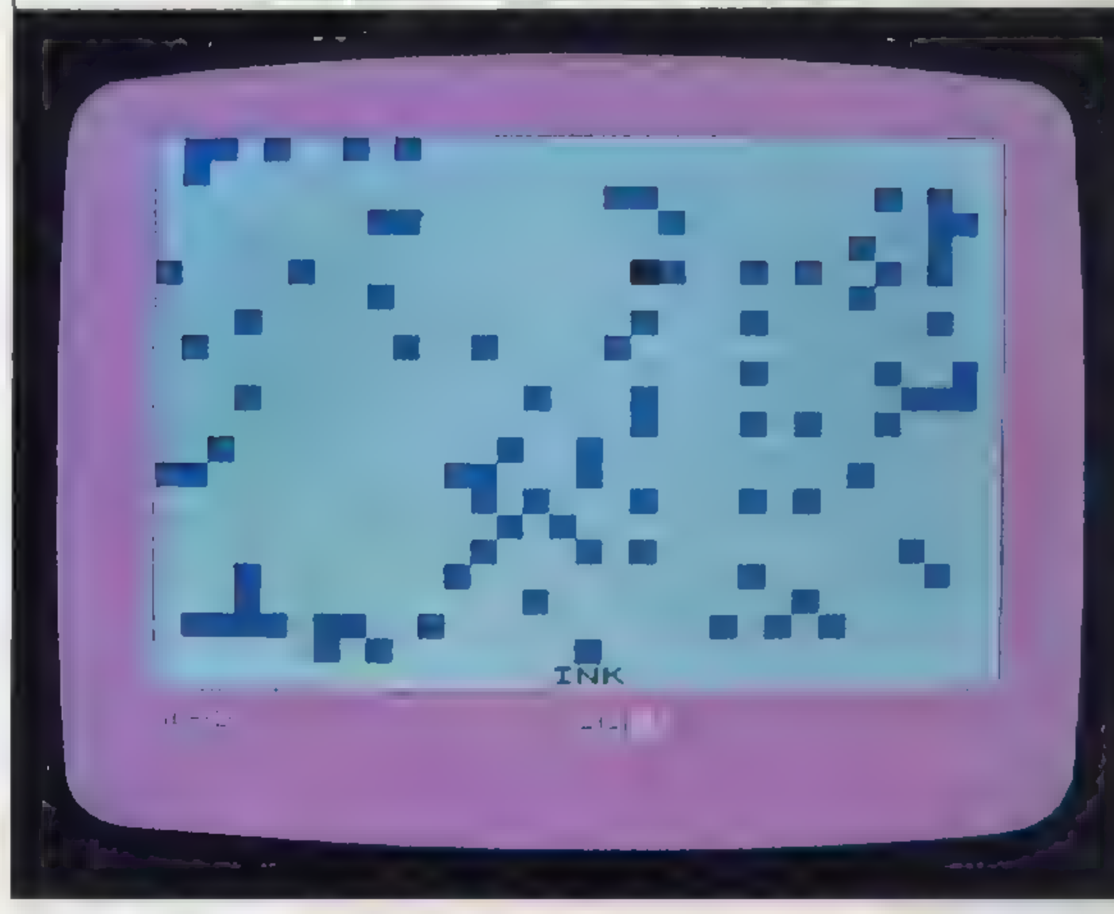
PAPER/INK TESTER



```
  10 BORDER 3. PAPER 5: INK 1. C
LS
  20 FOR n=1 TO 100
  30 LET r=INT (RND*21)
  40 LET c=INT (RND*32)
  50 PRINT AT r,c."■"
  60 NEXT n
  70 INK 0
  80 INPUT x,y
  90 LET a=POINT (x,y)
 100 PLOT x,y
 110 IF a=0 THEN PRINT AT 21,14;
"PAPER"
 120 IF a=1 THEN PRINT AT 21,14.
" INK "
 130 GO TO 80

0 OK, 0.1
```



PAPER/INK TESTER DISPLAY



Lines 20 to 60 PRINT dark blue squares in random positions on a cyan background. The bottom line of the screen is left blank for use later. Line 70 switches to black INK. Line 80 waits for you to type in the x and y co-ordinates of any point on the screen.

Type in an x co-ordinate (0–255), press ENTER, and then type in the y co-ordinate (0–176) and press ENTER again. Line 90 will then test whether this point is PAPER or INK. Line 100 PLOTs a black point at x,y to mark it. Note that this program would not work if lines 90 and 100 were reversed, because x,y would always be an INK colour, coming immediately after PLOT x,y.

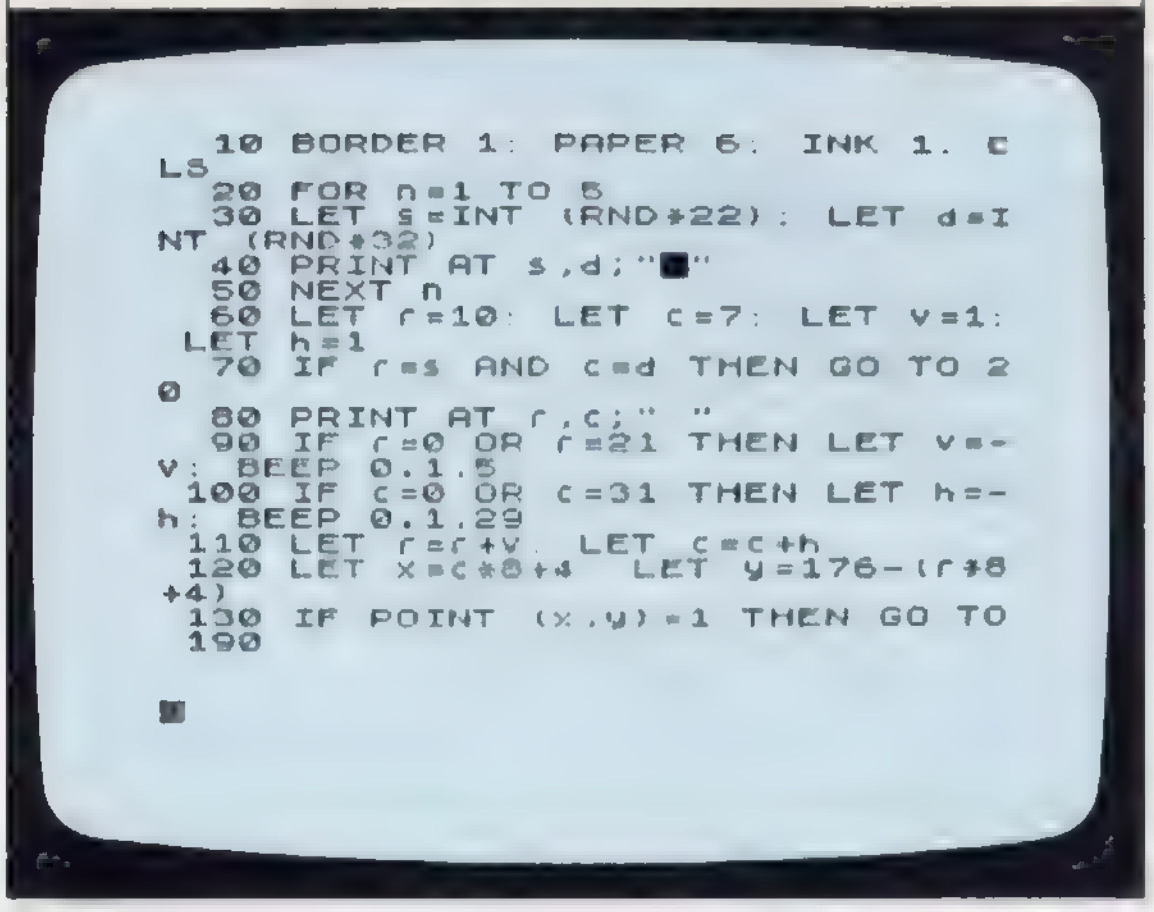
The result of line 90 is PRINTed on row 21 by lines 110 and 120. If x,y is a PAPER colour, the point ENTERed appears as a small black dot on the screen. If x,y is an INK colour, the whole character-sized square turns black because if you change the INK colour of a single point in any character position, the whole character position changes to the new INK colour.

Using POINT with animation

Now you know how POINT is used, you can try out a practical application of the command. You have probably come across a situation where you have several characters on the screen, but you don't want to go to the trouble of storing their positions and recalculating when any of the characters move in order to keep track of them during animation.

The next program gets around that problem. It uses POINT to test for the position of obstacles scattered over the screen as a character bounces between the screen edges. There are only a few obstacles in this display, but you can easily increase the number by altering the range of the loop between lines 20 and 50:

MINESWEEPER PROGRAM



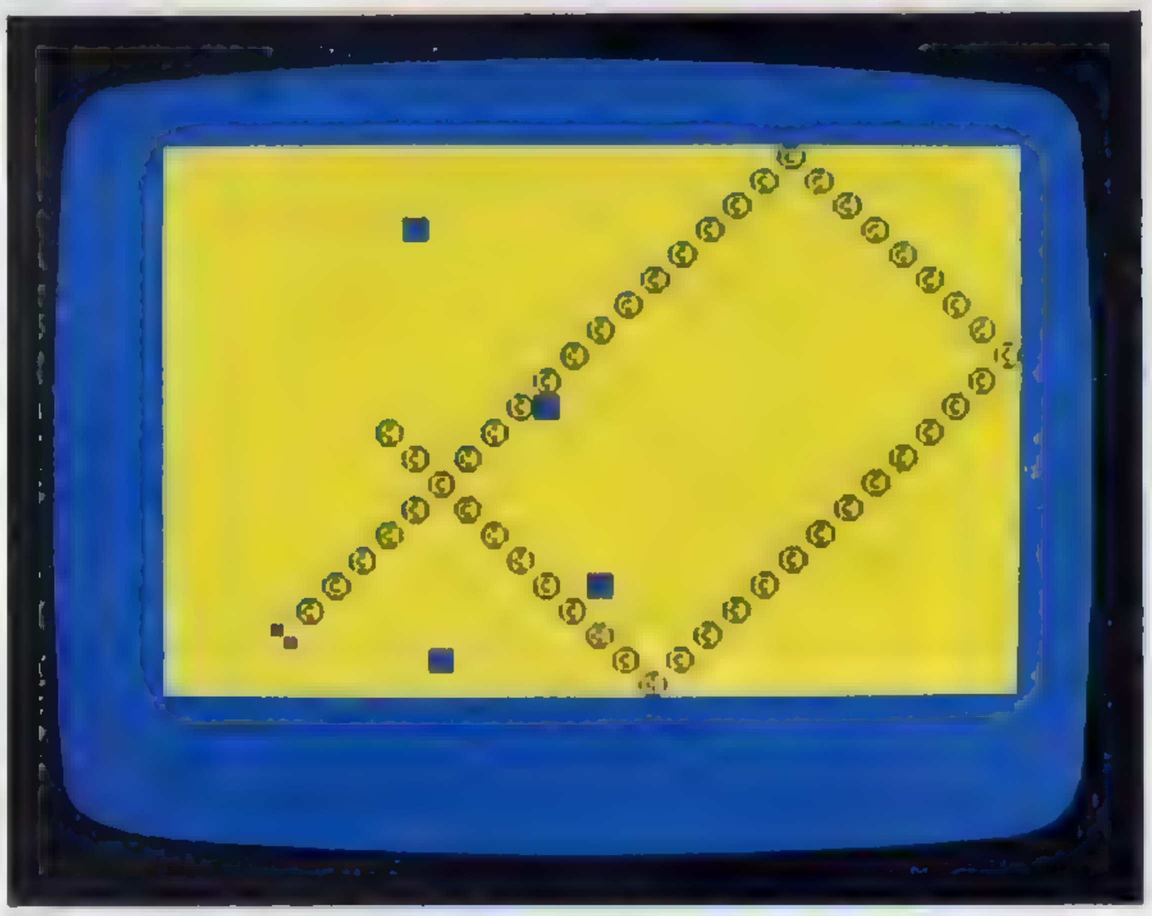
```
  10 BORDER 1: PAPER 6: INK 1: C
LS
  20 FOR n=1 TO 5
  30 LET s=INT (RND*22): LET d=I
NT (RND*32)
  40 PRINT AT s,d;"■"
  50 NEXT n
  60 LET r=10: LET c=7: LET v=1:
 LET h=1
  70 IF r=s AND c=d THEN GO TO 2
0
  80 PRINT AT r,c;" "
  90 IF r=0 OR r=21 THEN LET v=-
v: BEEP 0.1,5
 100 IF c=0 OR c=31 THEN LET h=-
h: BEEP 0.1,29
 110 LET r=r+v: LET c=c+h
 120 LET x=c*8+4: LET y=176-(r*8
+4)
 130 IF POINT (x,y)=1 THEN GO TO
 190
```

```
 140 PRINT INK 0;AT r,c;"©": PAU
SE 3
 150 GO TO 80
 160 FOR n=1 TO 10
 170 FOR a=1 TO 6
 180 PRINT INK a;AT r,c;"▚": BEE
P 0.05,10
 190 LET a=a+1
 200 PRINT INK a;AT r,c;"▞": BEE
P 0.05,30
 210 NEXT a
 220 NEXT n
 230 GO TO 10

0 OK, 0:1
```

The program uses POINT to locate INK characters on a PAPER screen. Lines 20 to 50 PRINT dark blue mines (squares) on a yellow PAPER screen. Line 60 sets the start position for the minesweeper (a copyright symbol). If this start position coincides with any of the random blue squares, a new set of random squares is PRINTed. Lines 80 to 150 form a routine that you have encountered several times now. It makes the minesweeper bounce around the screen from side to side and top to bottom. Every time the minesweeper reaches one of the screen's four edges, its direction is reversed and a BEEP sounded. The BEEP produced when the minesweeper reaches the sides of the screen is much higher in pitch than when it reaches the top or bottom edge.

For each position the minesweeper symbol moves into, line 120 converts its row and column co-ordinates into the x,y graphics co-ordinates of the mid-point of the position. Line 130 then checks whether this is a PAPER or INK colour. If it is a PAPER colour, the program jumps back to line 80, erases the minesweeper and rePRINTs it in a new position (calculated by line 110). If it's an INK colour, line 130 returns the value 1 and the program jumps to the explosion subroutine at line 190. This repeatedly PRINTs two graphics characters in a range of different colours, accompanied by sound effects. After every explosion, the program jumps back to line 10 and starts again:

MINESWEEPER DISPLAYS



The number of mines is set to 5 so that the minesweeper has enough space to go back and forth across the screen several times before it hits anything. You can provoke an explosion more quickly by increasing this number in line 20. Masking the PAUSE in line 140 will also speed up the program.

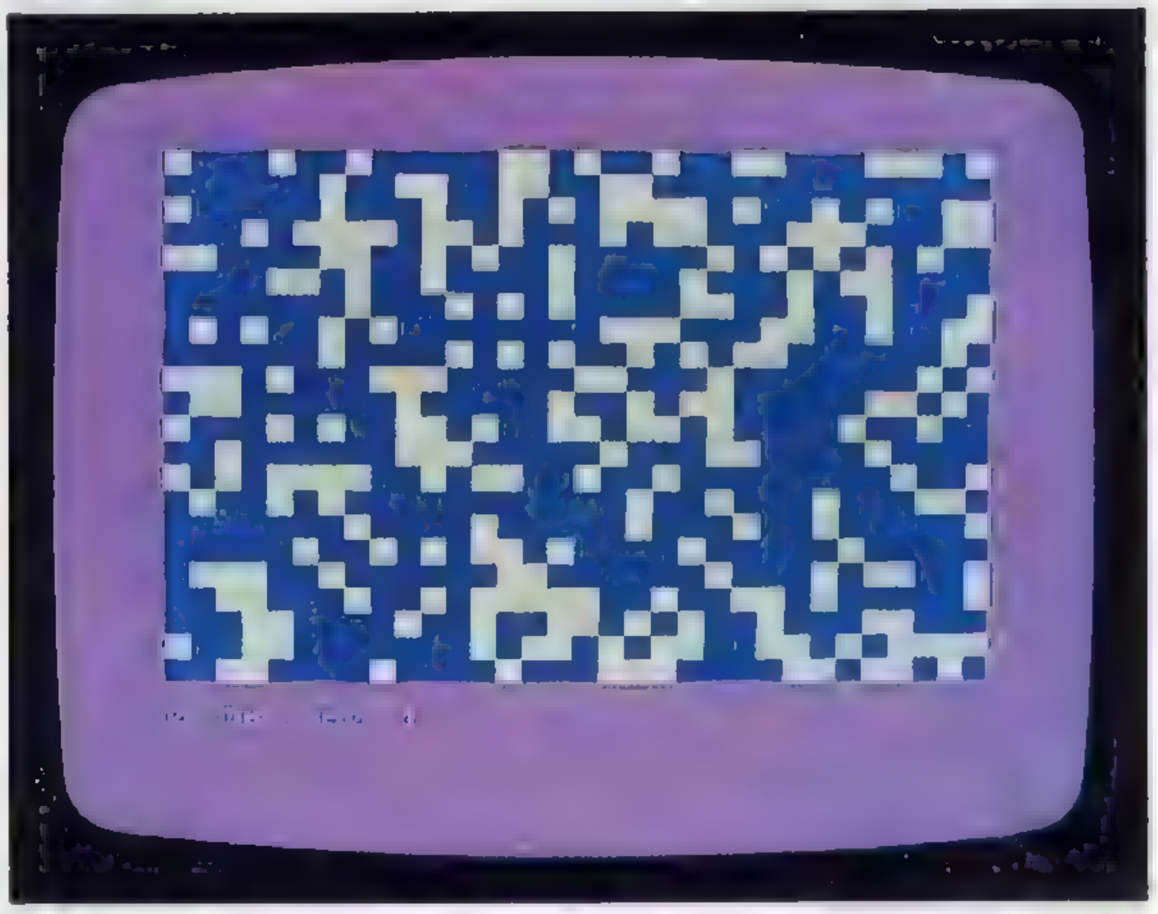
Using POINT in routines like this enables you to program the computer to make a complex decision with a simple series of commands. POINT is very useful in games where you are controlling an object that has to be steered around the screen and kept clear of walls or other obstacles. You can test your skill at computerized navigation by using POINT to direct the program into a crash or penalty subroutine when you steer into INK.

PATTERNS WITH SYMMETRY

If you make the Spectrum produce a pattern at random, the display produced will be entirely unpredictable. Using a simple technique, it is possible to generate a random display in a way that produces a symmetrical result. An unpredictable display makes a random pattern all over the screen, a technique that is useful for producing a background "galaxy" effect for a star wars game. But with a symmetrical display, the first part of the program produces a random pattern in one quarter of the screen, while subsequent parts of the program repeat this random pattern in each of the other three corners of the screen. The effect is like seeing part of the display reflected in a mirror.

To see how you do this, first you need a program that produces a random display. On the Spectrum this is very easy:

RANDOM DISPLAY PROGRAM



This short program repeatedly produces row and column co-ordinates for a random location on the screen. Although the column furthest to the right is numbered 31, the expression in line 30 that produces the column number includes 32, because INT rounds numbers down to the nearest integer, so the highest value that INT(RND*32) can have is 31. The graphics symbol on key 8 is used to produce the display. The program positions 300 squares at random. There are a total of 704 character positions, so, at most, about half the screen is covered. There's nothing to prevent lines 30 and 40 producing the same pair of co-ordinates on several occasions, so that often the INK squares will take up a smaller area than this.

Reflecting a random pattern

Now you can use the graphics symbol, but in a slightly different way, to turn this entirely random pattern into a random pattern with symmetry:

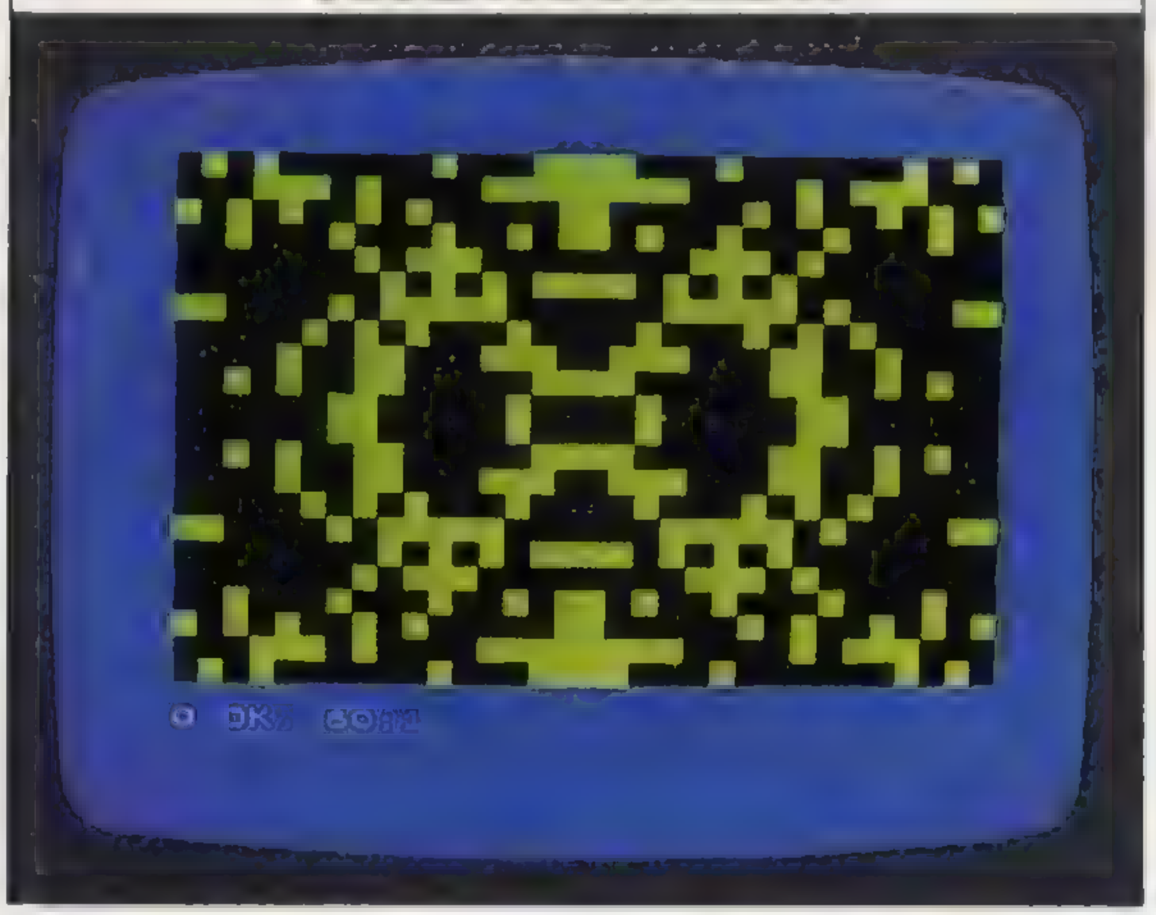
SYMMETRICAL DISPLAY PROGRAM

This program divides the screen into four equal quarters. All of the co-ordinates that lines 30 and 40 produce lie in the top left-hand quarter. They are copied onto corresponding locations in the other three quarters. Because each pair of co-ordinates produced gives rise to four symbols PRINTed on the screen, the program only needs to carry out a quarter the number of loops as the previous program. The maximum value of n in line 20 is therefore reduced to 75, but this produces 300 images as before. The first of the next pair of screens shows a typical RUN of the program. The second screen shows the result of adapting the program by changing line 10 to:

```
10 BORDER 1:PAPER 0:INK 6:CLS
```

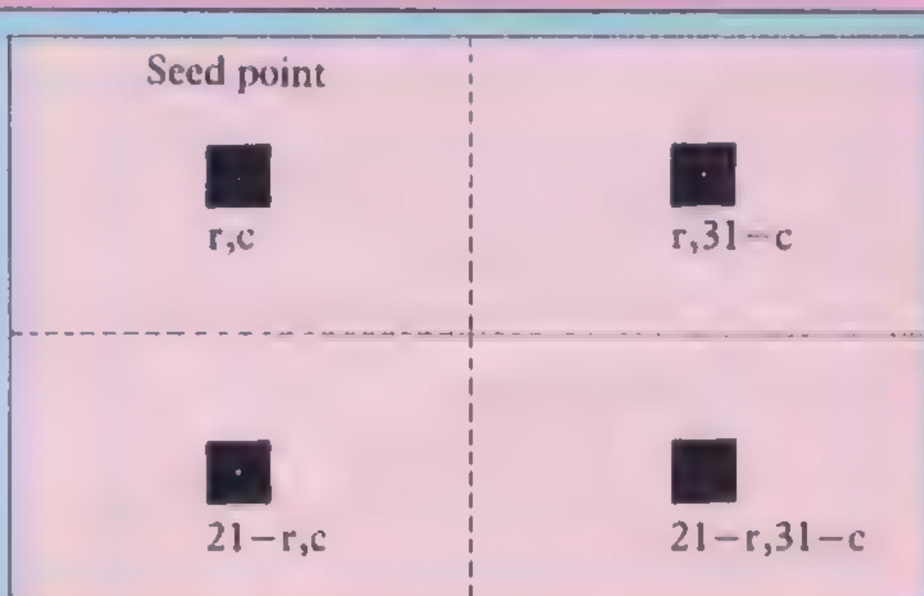
and then using a smaller graphics character (this time on the 3 key). The program will produce a different display every time it is RUN:

SYMMETRICAL DISPLAYS



The program works out the positions of the mirror images of each original "seed" symbol by performing some simple arithmetic on the co-ordinates r,c. The first mirror image is on the same row as the seed, but on the opposite side of the screen. Its co-ordinates are therefore r,31−c. Each point is reflected by two others in the bottom half of the screen.

USING SYMMETRICAL CO-ORDINATES

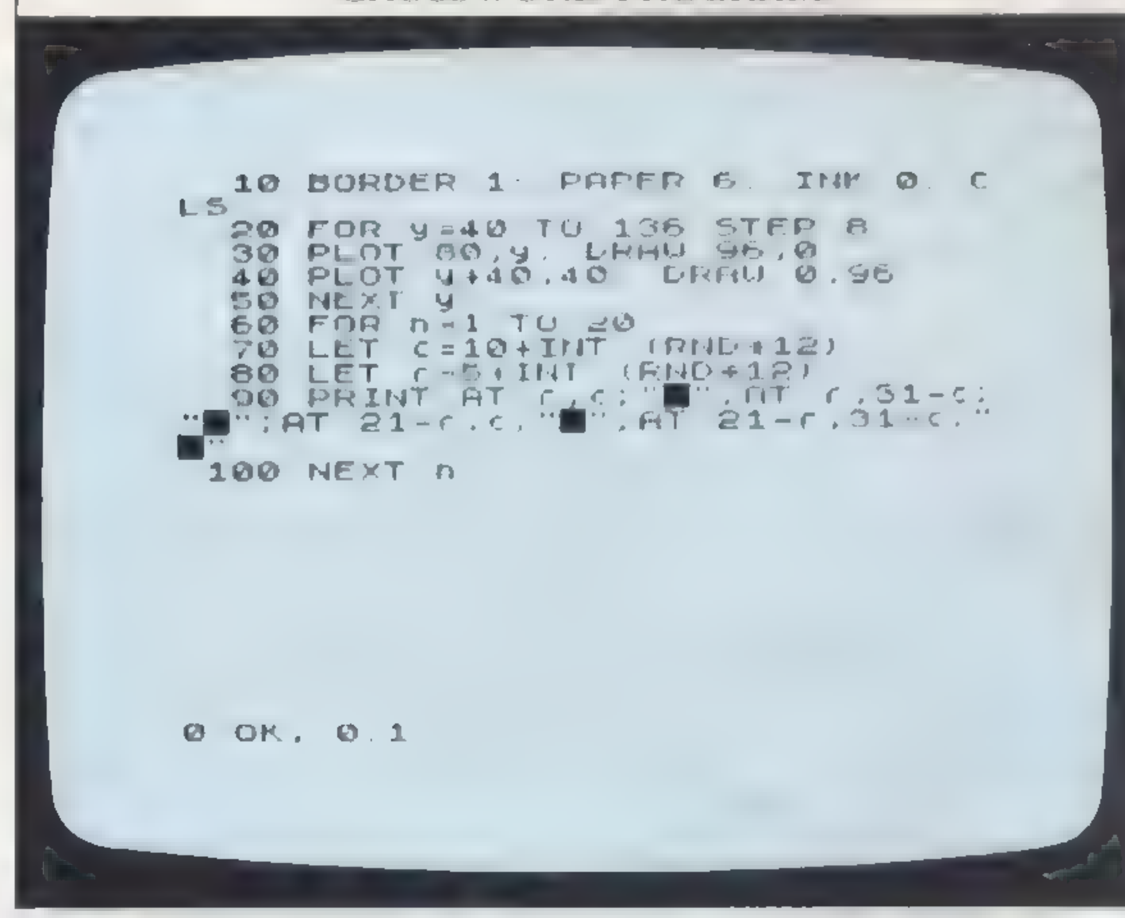


Each is as far up from the bottom of the screen as the seed point is down from the top, so their co-ordinates are the 21−r,c and 21−r, 31−c in line 50.

Programming a crossword grid

Most crosswords are constructed from a symmetrical grid, and by reflecting the black squares in one corner of the crossword grid, you can program the computer to PRINT the complete pattern. Again, "seed" squares are positioned at random:

CROSSWORD PROGRAM



Lines 20 to 50 DRAW the grid. The grid is 12 squares across and 12 squares down. This line spacing is chosen so that each blank square that lies within the grid is the same height and width as a keyboard character (8 graphics pixels across and 8 down). The RND statements in lines 70 and 80 are chosen so that the random co-ordinates produced by them always lie within the crossword grid. Line 90 then PRINTs the seed point and its three mirror images as in the previous program. Wherever a graphic square is PRINTed, it fills a blank square in the grid.

TRACING ERRORS

The Spectrum is much more helpful than many microcomputers in tracing errors or "bugs" in programs. Some computers will let you type in almost anything; you only discover that your program does not make sense when you RUN it and get a whole succession of error reports. The Spectrum, on the other hand, checks each line you type in for errors before you can use it in a program.

Although you cannot spell a keyword wrongly, because the Spectrum's entry system works on single keys, it is easy to use punctuation incorrectly. If you miss out a semi-colon after an AT, or separate two commands with something other than a colon, for example, a flashing question mark appears on the line when you press the ENTER key. Its position indicates where the error is. All you have to do is move the cursor back down the line and correct the error. If the cause of the error isn't immediately apparent, check that you have used the commands in question correctly.

This system of entry-checking does not mean that you can't make a mistake when programming the Spectrum. You can easily write a program which the Spectrum will accept, but which is still riddled with errors. Here is a program that is full of errors. If you key it in (the computer will accept all the lines) you can then see how to go about a thorough "debug".

The program is the "hangman" game from page 27, but it has been written and ENTERed hurriedly, so that it will not work. Don't cheat by looking back at the correct program! Go through the listing and see if any bugs are obvious to you. See how many you can find, and try and work out how you would correct them. Then check your results against the bugs that are explained on these pages.

Key in the program as it is and then try to use it:

BUGGED "HANGMAN" PROGRAM

```
  10 BORDER 0  CLS   PRINT AT 1,
12;"HANGMAN"
  20 PRINT AT 10,2;"Ask a friend
 to type a word"
  30 PRINT AT 12,3;"or phrase fo
r you to guess"
  40 PRINT AT 18,4;"DON'T LOOK A
T THE SCREEN"
  50 PRINT AT 20,0;"Press ENTER
when you're finished"
  60 INPUT a$
  70 LET l=LEN a$
  80 FOR n=1 TO l
  90 IF a$(n)=" " THEN PRINT AT
11,(32-l)/2+n;"■"
 100 PRINT AT 11,(32-l)+n,"J": N
EXT l
 110 PRINT AT 2.12;"HANGMAN";AT
5,6;"-:letter    ■ space"
 120 PRINT AT 18,6,"     Try a le
tter    ";AT 19,0;"Press 1 to gu
ess the whole thing"

scroll?
```

```
 130 INPUT t$
 140 IF t$="2" THEN STOP
 150 IF t$="1" THEN GO TO 190
 160 PRINT AT 16,12;"score=";s
 170 FOR n=1 TO l
 180 IF t$=a$(n) THEN PRINT AT 1
1,(32-l)+n;t$
 190 NEXT n
 200 GO TO 130
 210 PRINT AT 18,6;"Try the whol
e thing". INPUT t$
 220 IF t$=a$ THEN PRINT AT 11,1
2;"CORRECT";AT 13.12,"score=";s+
1
 230 GO TO 120

0 OK, 0:1
```

When you try to RUN the program you will find that the title frame comes up and you are invited to ENTER the test string. But when your press ENTER, the error report "1 NEXT without FOR, 100:2" appears. This means that the program has stopped at the second statement in line 100. LIST the program. You will see that a loop containing n begins at line 80, but the NEXT statement in line 100 says NEXT 1. To make the loop work properly, change this to NEXT n. Now try the program again:

"CRASHED" PROGRAM DISPLAY

```
              HANGMAN




   Ask a friend to type a word
                         JJJJJJJ
    or phrase for you to guess




     DON'T LOOK AT THE SCREEN
Press ENTER when you're finished
```

This time the title frame and test string entry work properly, but the title frame stays on when the next phase of the game starts. That's easily dealt with. Add :CLS to line 70. You may also have got a "B Integer out of range" error report on screen. The characters representing the test string are also PRINTed in the wrong position. They may have run out of the side of the screen as shown above.

The expression in line 100 should PRINT all the characters in the middle of the screen. It should read:

(32−1)/2+n

but in the program, the division by 2 has been missed out altogether.

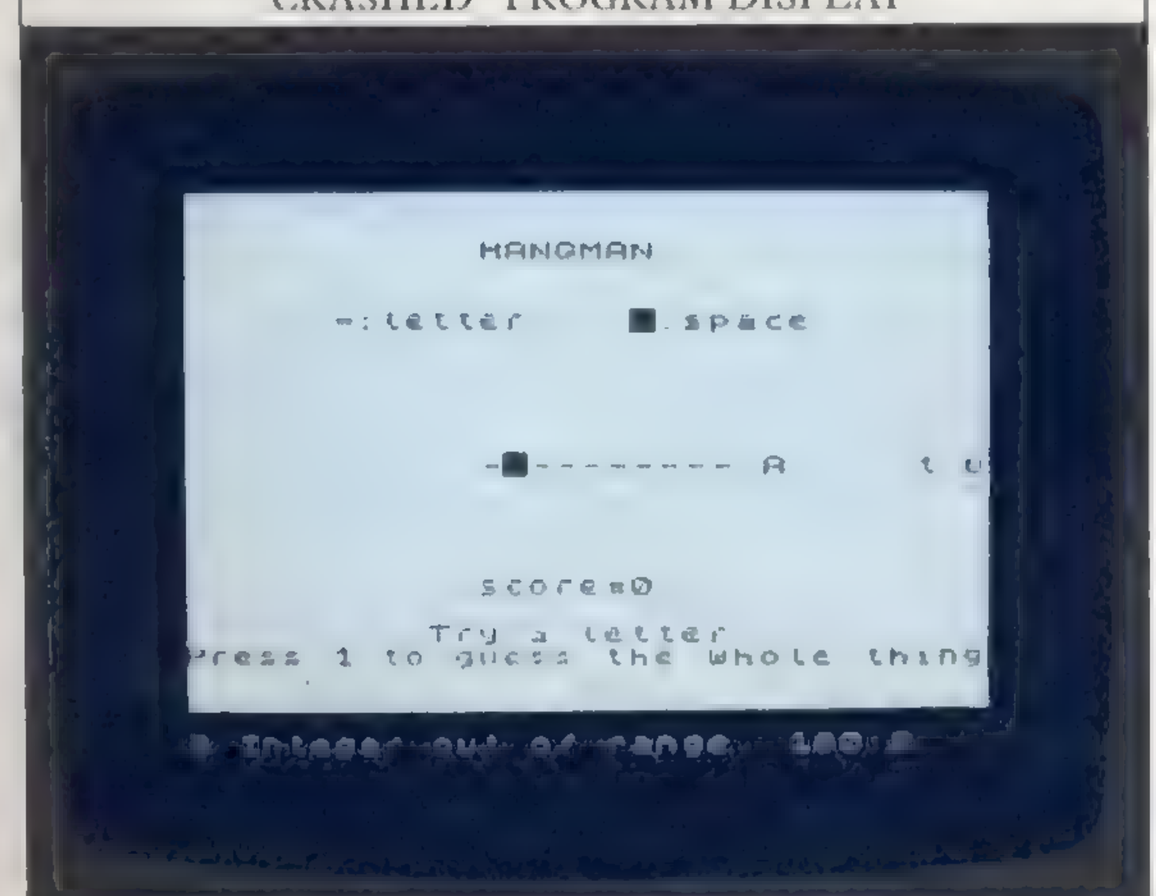
The display instructions tell you that a hyphen represents a letter, but a row of the letter j is PRINTed instead. Moreover, if you've ENTERed a test string containing any spaces, you'll have found that the black square graphics symbol has not appeared.

The j-line has appeared because the programmer forgot to press the shift key to get a hyphen. Correct that, then look at line 90 again. This line checks for a space in the test string and if necessary PRINTs a black square, but this is masked by whatever line 100 PRINTs. If you add: NEXT n to the end of line 90, then when a black square is PRINTed, the program will move on to the next value of n and the next character in the test string.

Further test RUNs

Now, when you RUN the program, as soon as you ENTER your first guess at a letter, you will get a "2 Variable not found, 160:1" error report. The only variable in line 160 is s, which represents the score. It seems straightforward – it PRINTs the score. This line is, in fact, correct. The error report is PRINTed because the computer doesn't have an initial value for s, which it can PRINT. So, add :LET s=0 to the end of line 70. When you have keyed in these corrections, RUN the program again:

"CRASHED" PROGRAM DISPLAY



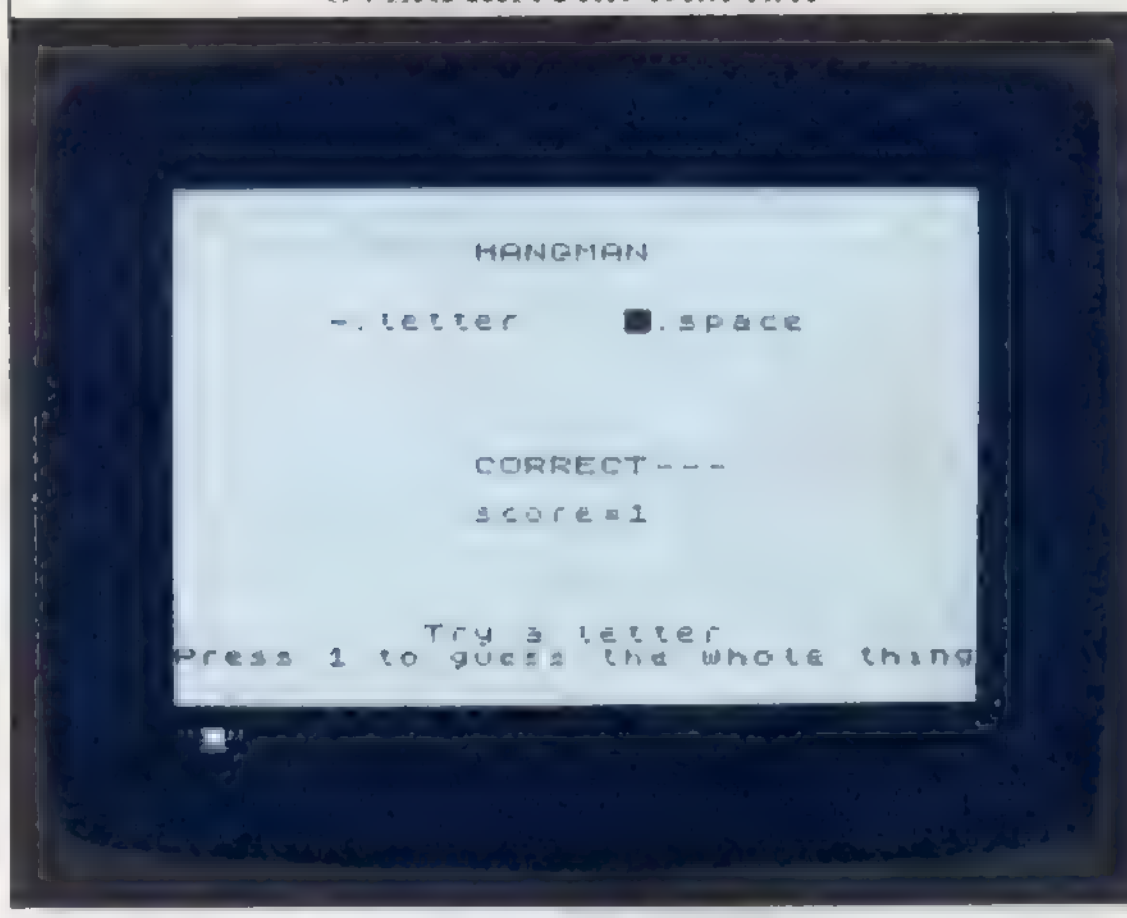
Now, when you type in a correct guess, it will probably appear to the right of the screen. Alternatively, if it represents a letter that appears towards the end of the test string, you may even get a "B Integer out of range, 180:2" error report, because the computer has tried to PRINT the character off the screen altogether. Once again, the division by 2 has been omitted from the column expression. It should read as follows:

(32−1)/2+n

The game now seems to work properly, but if you look at the score, you will see that it's not increasing. The score should go up by 1 after each guess and this has not been written into the program, so add LET s=s+1: to the beginning of line 160.

Although single-letter guesses will work properly now, the computer refuses to respond correctly when you press 1 to guess the whole string. Line 150 should make the program jump to the whole word guess routine, but it actually leads to line 190, which is simply NEXT n. This is a very common fault after you have renumbered any of the program's lines, and perhaps forgotten to renumber the GOTOs and GOSUBs. Change the GOTO statement in line 150 to GOTO 210. The following display should appear:

OVERPRINTED DISPLAY



Try typing in a correct guess for the whole string. The "CORRECT" frame is PRINTed over the previous game frame. That's easily put right – change line 220 to:

```
220 IF t$=a$ THEN CLS:PRINT AT 11,12;
    "CORRECT"; AT 13,12;"score=";s+1
```

Finally, the program refuses to stop. When you've made a correct guess, the program immediately restarts. So, add :STOP to the end of line 220. The program should now RUN properly.

If you are developing a program, constant checking should prevent all but a few bugs from slipping into the final listing. When you're testing a program that you've written, put it through all the situations it will meet in use. If it's supposed to have safeguards to stop it "crashing" in some circumstances, test them too. Testing the end of a growing program with GOTO will ensure that each routine actually works before you move on to the next.

SPEEDING UP PROGRAMS

One of the features of BASIC is that, if you want to, you can write programs without much prior planning. With many other computer languages, a much more organized approach is needed. Although BASIC is easy to use, it doesn't encourage the best programming.

One area where good and bad programming show up is in RUNning speed. While it's not of great importance that a short program RUNs quickly, RUNning speed does become more significant as your programs become longer and more complex. Ideally, you will want them to RUN as quickly as possible. On these two pages you can see how a poorly written program compares with a version that has been streamlined.

A "slow" program

The following program is of the type used on page 23 to show how arrays can produce spreadsheets on the screen. Here is the program listing:

"SLOW" TAX TABLE PROGRAM

```
  10 REM Tax table
  20 DEF FN t()=(65536*PEEK 2367
4+256*PEEK 23673+PEEK 23672)/50
  30 LET T=FN t()
  40 DIM p(8): DIM n(8)
  50 DATA 1.20,2.30,1.10,2.00,1.
80,2.90,1.45,2.05
  60 DATA 14,16,24,17,15,11,14,2
0
  70 FOR a=1 TO 8
  80 READ p
  90 LET p(a)=p
 100 NEXT a
 110 FOR a=1 TO 8
 120 READ n
 130 LET n(a)=n
 140 NEXT a
 150 CLS
 160 FOR y=20 TO 164 STEP 16
 170 PLOT 28,y: DRAW 200,0
 180 NEXT y

scroll?
```

```
 190 PLOT 28,20: DRAW 0,144: PLO
T 68,20: DRAW 0,144: PLOT 92,20:
 DRAW 0,144
 200 PLOT 140,20: DRAW 0,144: PL
OT 180,20: DRAW 0,144
 210 PLOT 228,20: DRAW 0,144
 220 PRINT AT 2,6;"£";AT 2,9;"No
";AT 2,13;"SUB";AT 2,18;"TAX";AT
 2,23;"TOTAL"
 230 FOR a=1 TO 8
 240 PRINT AT 2*a+2,4;p(a)
 250 PRINT AT 2*a+2,9;n(a)
 260 PRINT AT 2*a+2,12;p(a)*n(a)
 270 PRINT AT 2*a+2,18; INT (15*p
(a)*n(a)+0.5)/100
 280 PRINT AT 2*a+2,23;p(a)*n(a)
+INT (15*p(a)*n(a)+0.5)/100
 290 NEXT a
 300 PRINT AT 21,4;"Time taken:
";FN t()-T

0 OK, 0:1
```

The program takes a list of unit prices, numbers of items and – given the current tax rate – calculates the total cost of the items. The information is presented as a table. To test the program's RUNning speed, lines 20 and 30 start timing the program as soon as it starts and line 300 PRINTs its total RUNning time on the bottom line of the screen:

"SLOW" TAX TABLE DISPLAY

£	No	SUB	TAX	TOTAL
1.2	14	16.8	2.52	19.32
2.3	16	36.8	5.52	42.32
1.1	24	26.4	3.96	30.36
2	17	34	5.1	39.1
1.8	15	27	4.05	31.05
2.9	11	31.9	4.79	36.69
1.45	14	20.3	3.04	23.34
2.05	20	41	6.15	47.15

```
Time taken 2.2600002

0 OK, 300:1
```

The unit prices are stored in the DATA statement at line 50, and the numbers of each item in line 60. Both arrays are dimensioned by line 40. The loop at lines 70 to 100 READs the unit price DATA into the one-dimensional, eight-element array p. A similar loop at lines 110 to 140 READs the numbers DATA into the n array. Lines 150 to 220 clear the screen, DRAW the grid of intersecting lines and PRINT the column headings – "No" for number and "SUB" for subtotal. Lines 230 to 290 calculate and PRINT the figures to fill the grid.

On looking through the listing, you might have noticed that two loops use the same variable (a) over the same range of values (1 to 8). It would have been possible to combine these loops and save space and time. Also the row number given by 2*a+2 is newly calculated in each line. This wastes some time. Finally, the same expression (p(a)*n(a)) appears three times in lines 270 to 290, calculated again each time it is required. All this shows that the program has been badly thought out, and needs improvement. It takes about two and a quarter seconds to RUN. Now see how much you can shave off this by using more economical programming techniques.

Techniques for time-saving

Here is the program again, but in a modified form. It produces exactly the same display, but in places the program statements are quite different:

IMPROVED TAX TABLE PROGRAM

```
  10 REM Fast Tax table
  20 DEF FN t()=(65536*PEEK 2367
4+256*PEEK 23673+PEEK 23672)/50
  30 LET T=FN t()
  40 DIM p(8): DIM n(8)
  50 DATA 1.20,14,2.30,16,1.10,2
4,2.00,17,1.80,15,2.90,11,1.45,1
4,2.05,20
  60 FOR a=1 TO 8
  70 READ p,n
  80 LET p(a)=p: LET n(a)=n
  90 NEXT a
 100 CLS
 110 FOR y=20 TO 164 STEP 16
 120 PLOT 28,y: DRAW 200,0
 130 NEXT y
 140 PLOT 28,20: DRAW 0,144: PLO
T 68,20: DRAW 0,144: PLOT 92,20:
 DRAW 0,144: PLOT 140,20: DRAW 0
,144: PLOT 180,20: DRAW 0,144: P
LOT 228,20: DRAW 0,144

scroll?
```

```
 150 PRINT AT 2,6;"£";AT 2,9;"No
";AT 2,13;"SUB";AT 2,18;"TAX";AT
 2,23;"TOTAL"
 160 FOR a=1 TO 8
 170 LET r=2*a+2: LET b=p(a)*n(a
): LET d=INT (15*b+0.5)/100: LET
 e=b+d
 180 PRINT AT r,4;p(a);AT r,9;n(
a);AT r,12;b;AT r,18;d;AT r,23;e
 190 NEXT a
 200 PRINT AT 21,4;"Time taken:
";FN t()-T

0 OK, 0:1
```

The first thing you will notice is that ten lines have been lopped off the listing. To achieve this, a lot of unnecessary calculations have been removed and several similar statements have been neatly condensed into a single line.

The two listings are identical up to line 50. All of the DATA has been written into this line, but it's also been reorganized. This is necessary because the loops that load the DATA into the two arrays have been rewritten. There is now a single loop at lines 60 to 90, saving four lines. Both the price and number are READ by READ p,n in line 70. The DATA in line 50 must, therefore, be presented in this way – a price followed by a number, and so on.

Lines 110 to 150 are identical to lines 160 to 220 in the first program, except that several of the PLOT and DRAW statements have been written into a single statement instead of three statements taking up three lines. The computer can deal with multiple statements in a single line substantially faster than with several separate statements, each occupying a different line.

There is now a major change in the way the grid is filled with figures. At the beginning of each loop, once the value of a has been established by line 160, all the calculations necessary for that loop are carried out once and for all by line 170. The variables used work in the following way:

TAX TABLE VARIABLES

The program uses four variables to fix positions and carry numbers calculated for use in the table.

Variable(s)	Function
r	Fixes row number where DATA will be PRINTed
b	Stores product of unit price and number of items (SUB column)
d	Stores amount of tax at 15% rate
e	Stores total cost (subtotal + tax)

Line 180 is now a lot less complicated, because all the calculations have already been done. It's just a matter of PRINTing the correct variable in the correct position in the grid. If you RUN the program in its improved form, the display now shows the time saved by making these changes:

IMPROVED TAX TABLE DISPLAY

£	No	SUB	TAX	TOTAL
1.2	14	16.8	2.52	19.32
2.3	16	36.8	5.52	42.32
1.1	24	26.4	3.96	30.36
2	17	34	5.1	39.1
1.8	15	27	4.05	31.05
2.9	11	31.9	4.79	36.69
1.45	14	20.3	3.05	23.35
2.05	20	41	6.15	47.15

```
Time taken  1.9799995
0 OK, 200:1
```

The changes have taken more than a quarter of a second off this short program, a saving of more than 12 per cent. You can imagine the saving in a longer program dealing with much more complicated numerical DATA or making lots of calculations for example.

When you are writing a long program, watch out for any repetition in the lines. The chances are that with some planning you could save both RUNning time and memory. Any large routine that is repeated a number of times will be worth making into a subroutine with GOSUB, while any calculations that crop up frequently may be carried out with FN. These devices, plus the various loops, can all be used to speed up a program. Because the speed with which your programs are RUN is slowed down by conversion into machine code, savings in BASIC are always worth considering.

HINTS AND TIPS

How to SAVE screen displays

Sometimes you may write a program which produces a display which you would like to be able to see again later. With the Spectrum you don't necessarily have to RUN the original program to do this. Instead, the direct commands:

```
SAVE "display" SCREEN$
```

can be used with a cassette recorder to store a picture called "display" – or any other name – on tape. The computer will then transmit to the tape the information about every pixel on the screen. When you want to see the image again, just play back the tape as usual, but with these commands:

```
LOAD "picture" SCREEN$
```

The computer will gradually scan across the television screen, PRINTing the pixels in exactly the same way as in the original display. By using this command, you can produce graphics directly without using program lines, but then store the result on tape cassette in exactly the same way that you would store a program.

Problems with functions

Two of the Spectrum's built-in functions, SQR and ↑, may sometimes cause problems in your programs. Because the square of any number is always positive, the function SQR cannot be used with a minus number, so if you are using SQR, make sure that this will not happen. The problem with the exponent function is less obvious. If you are mathematically-minded, you might have tried the following way of producing a number that is either +1 or −1 (this can be used in a program to produce lines or characters over the screen in an unpredictable way):

RANDOM GRAPHICS WITH EXPONENTS

The exponent function should produce a positive or a negative number, either +1 or −1 each time. In fact, it won't work. This is not because the maths is faulty, but because, like SQR, the Spectrum's exponent function will not work with a minus number. The answer will always come out positive – not much help for up-or-down movement. The best way to produce the effect is to set up a decimal figure from 0 to 0.99999999 with RND, and then use an IF … THEN line to produce +1 or −1 like this:

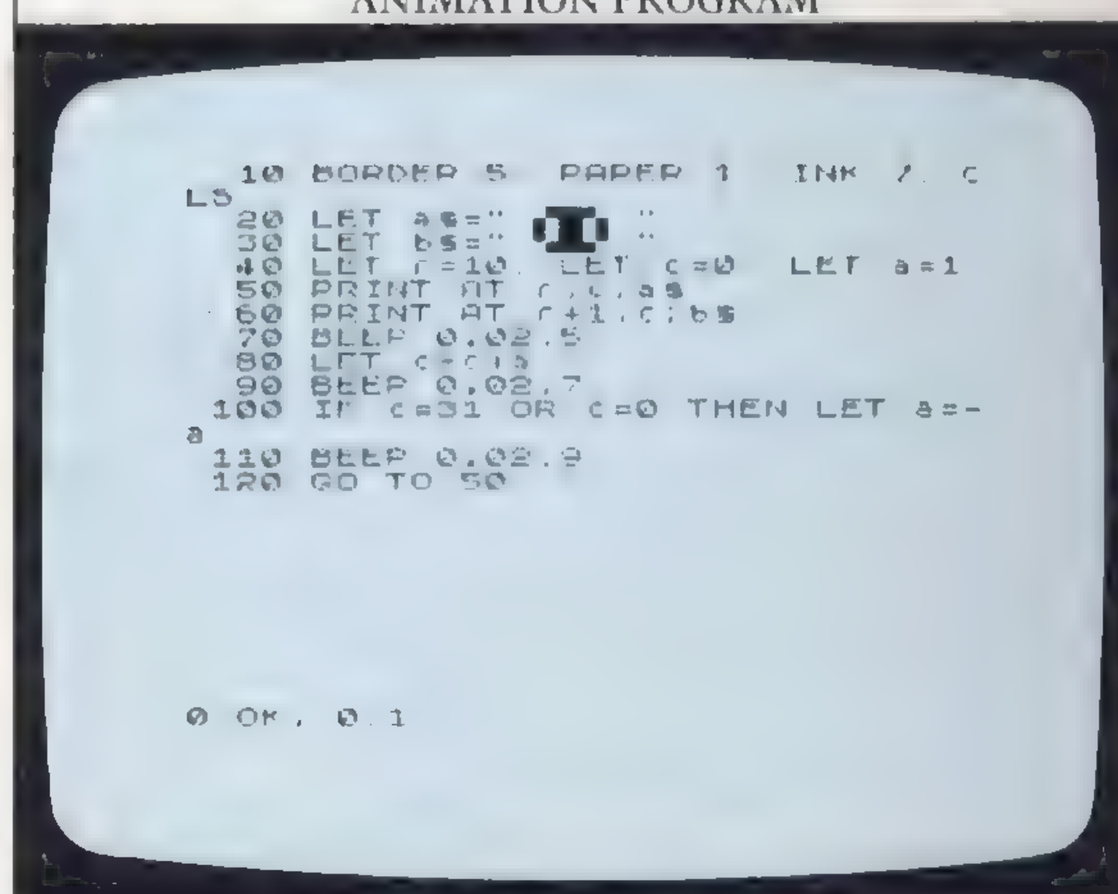
```
200 LET a=RND
210 IF a>0.5 THEN LET a=1: IF a<=0.5 THEN
    LET a=-1
```

SQR used with a minus number will produce an error report. The exponent function with a minus number will not, so you may not realize at first why a program is RUNning oddly.

Setting boundaries with animation

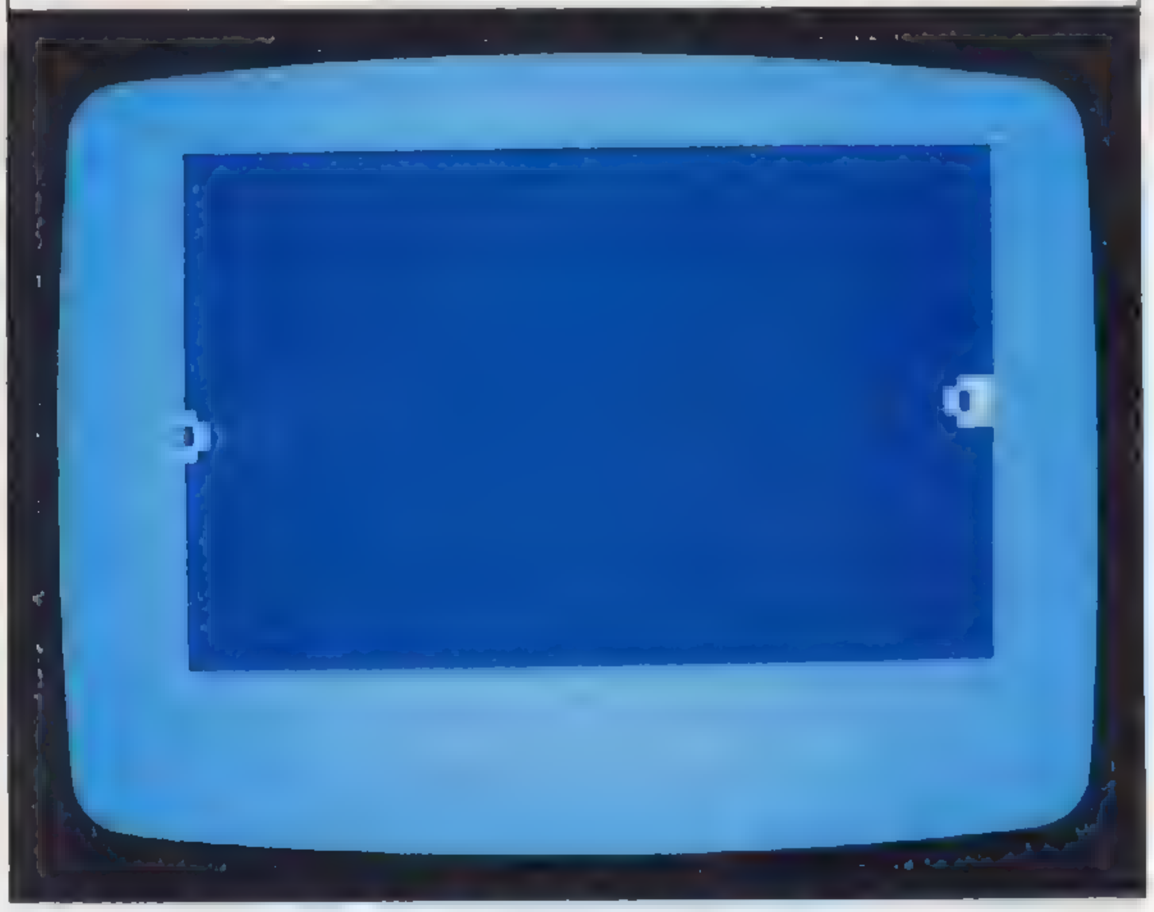
When you start animating graphics, you may find that they don't behave as you wish when they reach the edge of the screen. Here is a program that demonstrates this problem:

ANIMATION PROGRAM



The program animates a small flying saucer symbol built up from two rows each of three characters with a space either side. The space erases the old image of the saucer as the new image is drawn one space to the left or right of the last position. Line 40 sets the starting position and sets a – the change in position of the saucer – to 1. Line 80 sets the position for the next PRINT). Line 100 checks whether the saucer has reached the edge of the screen (c=0 or c=31) and if so, it reverses the saucer's direction. Then back to line 50 to PRINT the new saucer:

SPLIT CHARACTER



But, as you can see, it doesn't work properly. The saucer appears to wrap round onto the next row down and then retreat back up. It's a common problem with animation programs.

Line 100 deals with how the saucer behaves at the screen's edges. Remember that c represents the column number of the first or furthest left character of the saucer. When the last saucer character (the furthest right) reaches the right edge of the screen, c has a value of 27, not 31. To rectify this, change line 100:

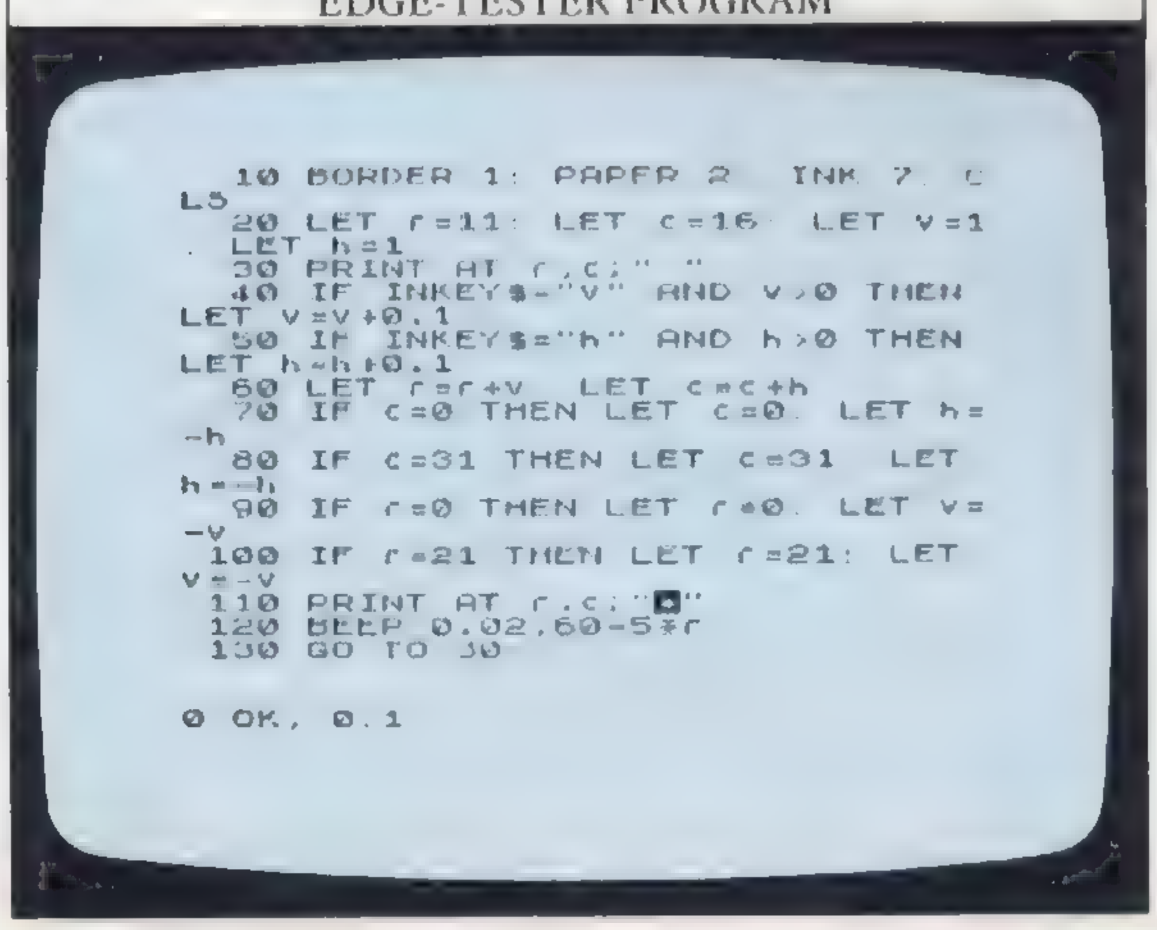
```
100 IF c=27 OR c=0 THEN LET a=-a
```

The program now takes into account the size of the object that is being animated.

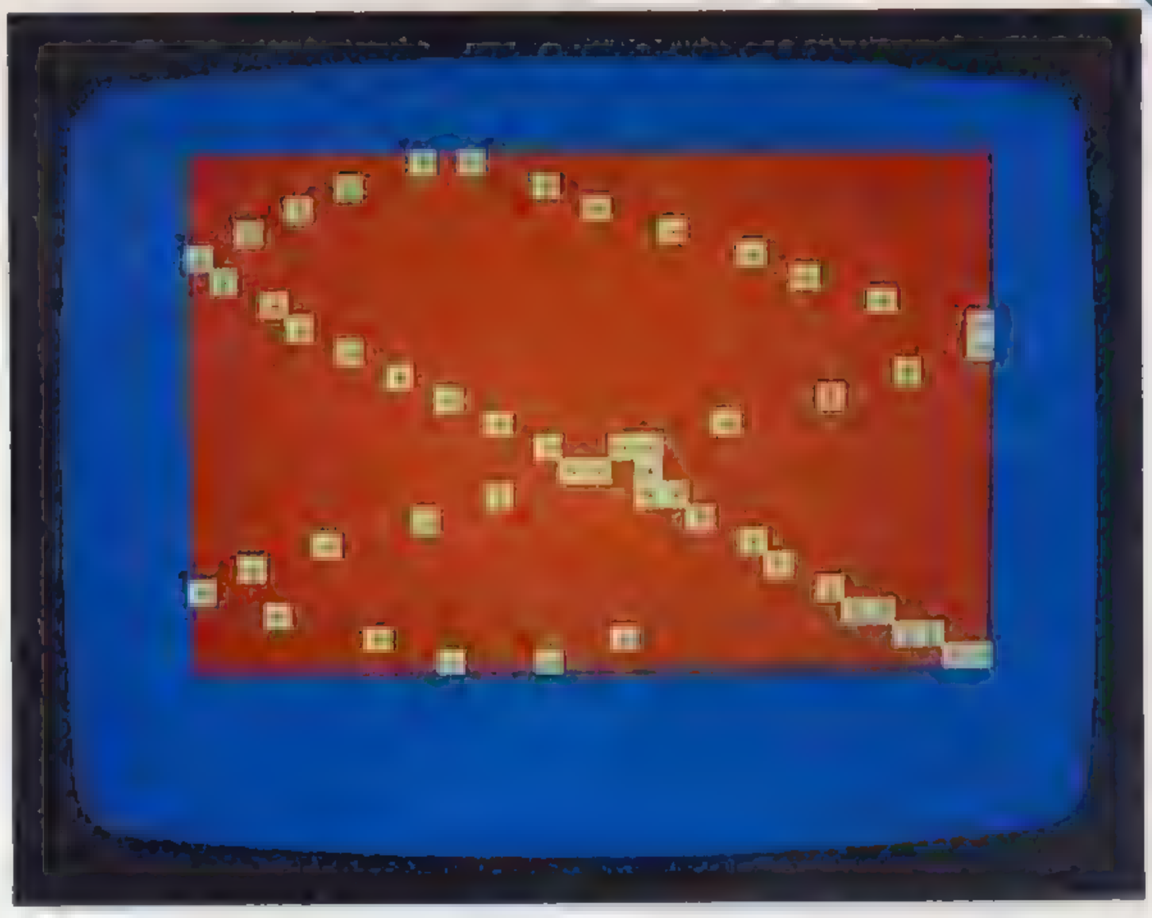
Testing for the screen edge

The next program is similar to the previous one in that it moves an object around the screen, but this time the object is composed of only one character and it moves in both horizontal and vertical directions:

EDGE-TESTER PROGRAM



```
  10 BORDER 1: PAPER 2: INK 7: CLS
  20 LET r=11: LET c=16: LET v=1: LET h=1
  30 PRINT AT r,c;" "
  40 IF INKEY$="v" AND v>0 THEN LET v=v+0.1
  50 IF INKEY$="h" AND h>0 THEN LET h=h+0.1
  60 LET r=r+v: LET c=c+h
  70 IF c=0 THEN LET c=0: LET h=-h
  80 IF c=31 THEN LET c=31: LET h=-h
  90 IF r=0 THEN LET r=0: LET v=-v
 100 IF r=21 THEN LET r=21: LET v=-v
 110 PRINT AT r,c;"■"
 120 BEEP 0.02,60-5*r
 130 GO TO 30

0 OK, 0.1
```



Each time the character is erased by line 30, line 60 changes the row and column values by v and h respectively. These are initially set to 1. Because of this and the effect of lines 70 and 80, the character bounces around the screen just like a ball bouncing around inside an empty box.

Lines 40 and 50 allow you to change the ball's speed. If you press the V key and the ball is travelling downwards, the variable v is increased by 0.1, increasing the ball's downward speed. In the same way, if you press the H key and the ball is travelling to the right, the ball's horizontal speed increases. But try it. When you press either V or H, the ball speeds towards the edge of the screen, and then the program throws up an error message – either "B Integer out of range, 110:1" or "5 Out of screen, 110:1". If you don't touch the keyboard, the program RUNs correctly, proving that the animation lines have been written properly, but for some reason will not work with INKEY$.

Consider for a moment what would happen if the ball was at position 20,10 and travelling downwards and to the right. At the same instant, you press the V key. Line 40 increases v from 1 to 1.1. Line 60 changes r to 21.1 and c to 11. Line 100 checks whether r equals 21. It doesn't. So, line 110 tries to PRINT the ball AT 21.1,11, producing the "Out of screen" error message.

The problem is that you can no longer predict the exact value of r or c when the ball reaches a screen edge. So, change some lines to cope with the situation and conditions you do know about – that is, neither r nor c can have values less than zero, r cannot be greater than 21 and c cannot be greater than 31. So the lines are now:

```
70 IF c<0 THEN LET c=0:LET h=-h
80 IF c>31 THEN LET c=31:LET h=-h
90 IF r<0 THEN LET r=0:LET v=-v
100 IF r>21 THEN LET r=21:LET v=-v
```

Once you have made these changes, the program will correctly test for the edge of the screen.

CONVERTING PROGRAMS

One of the problems with BASIC is that it exists in many different forms or "dialects". The Spectrum uses a form of BASIC which includes some commands – BORDER, PAPER and INK for example – that other machines do not have. Similarly, other micros may produce effects that can be achieved on the Spectrum, but by completely different techniques; nowhere is this more evident than in graphics programs. All this makes "program mobility" – trying to use the same program on different computers – very difficult. You can see this in software retailing where many programs are only available for one type of machine.

If you do see a program in a magazine or book, or a friend shows you a program RUNning on another computer, you may decide that it's worth converting to RUN on your Spectrum. To do this you must know not only how the machines concerned differ, but also you must be able to understand why and how a program does what it does. Then you can break the program down into blocks or subroutines and finally look at it line by line. But in the same way as you can rarely translate a message into a foreign language word for word, you cannot simply translate each "foreign" program statement directly into the equivalent statement for your computer. It may be more economical in time and more efficient to completely rewrite a section of program using the best commands available on your machine.

Points to watch for when converting

One of the most variable aspects of BASIC is punctuation and spacing. The Spectrum's error-checking system will ensure that you do not ENTER any lines that have incorrect spelling or punctuation, but you don't want to have to keep experimenting until you get things right. Full stops, colons and semi-colons are used in different ways by many machines, and you will often need to make punctuation changes before you can use lines on your Spectrum.

Remember that any text or graphics co-ordinates are likely to need changing. Because different computers often have different display resolutions, the co-ordinates used to produce displays can rarely be incorporated without alterations in converted programs. Making a note of the other micro's text and graphics grid limits will help you here.

When converting programs, you must always be on the lookout for commands that relate directly to a machine's operating system, because they will have no meaning for the Spectrum. Similarly, commands like PEEK and POKE on the Spectrum cannot be used in a program converted for other machines, as they refer to the Spectrum's memory addresses. Watch out too for commands that use a computer clock: you will need to completely rewrite routines to make use of the Spectrum's timing system.

The list of these problems is quite a long one, but once you have tried some program conversion, you will soon learn what is safe "standard" BASIC, and what is "dialect". It is a good idea when converting programs to have a look through the other computer's manual, so you can pick out any commands which look unfamiliar. If you know what these do in advance, you will find the process of translating a lot easier.

Converted and original listings

The following listing is an example of a program written specifically for another computer, the Acorn BBC Micro. It sets up a games board – a routine that you might want to use on your Spectrum. But it won't RUN on the Spectrum in its present form:

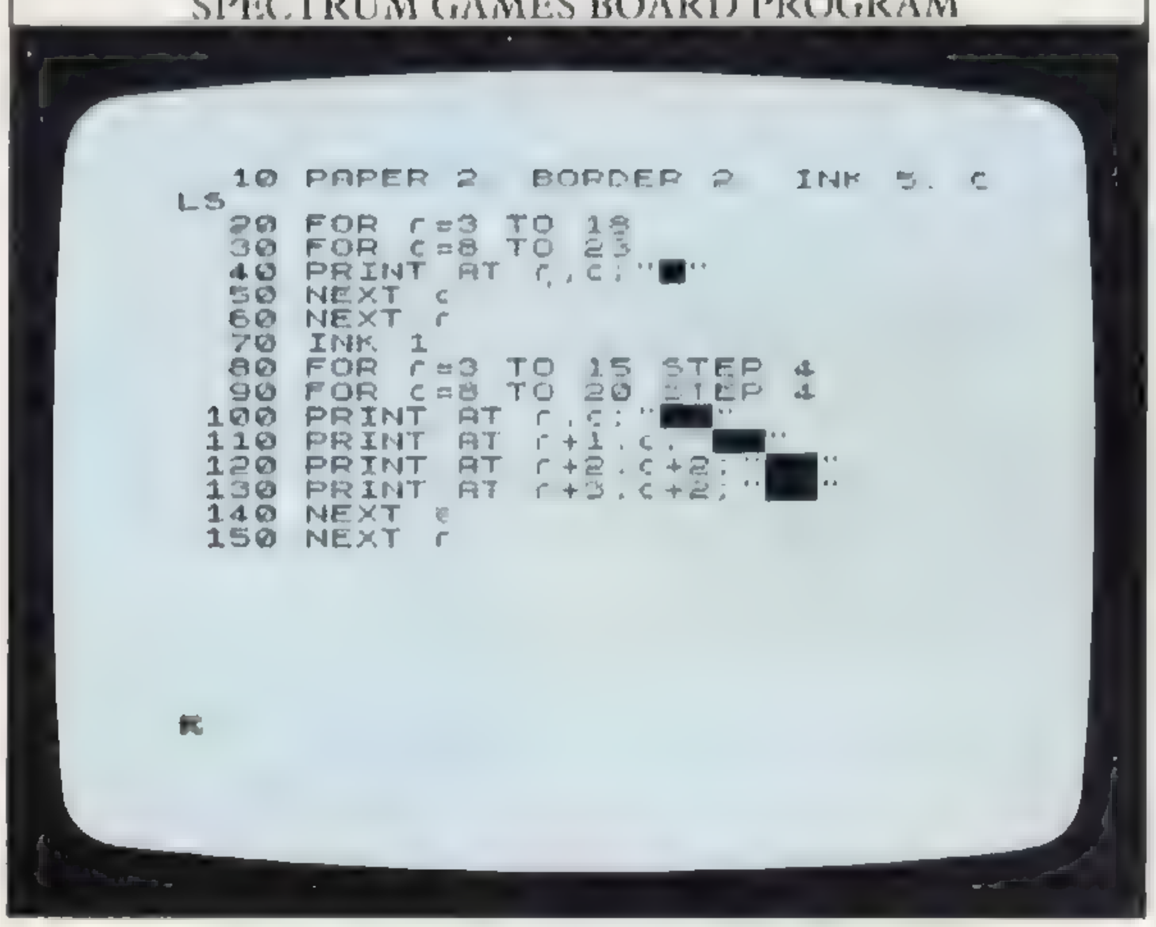
BBC MICRO GAMES BOARD PROGRAM

Almost everthing in this program is peculiar to the BBC Micro and cannot be used by the Spectrum. The BBC Micro's colour commands and the way in which it calls special kinds of subroutines, called procedures, are entirely different. That's why it is necessary to understand how the program works before you can start converting it. You will find that it is a great help if you can see the program RUNning rather than working only from a printed listing.

The most straightforward way to convert or translate this program is to take each block, identify its function, and then write a routine that will carry out the same function in Spectrum BASIC. Lines 10 to 110 form the main program, which jumps to two subroutines called PROCSCREEN and PROCBOARD. These are called by name instead of line number. PROCSCREEN draws a large cyan square on a red background. The main program draws a row of alternately coloured squares across the screen (lines 50 and 60) then moves down (line 70) and draws a second row of squares. The second row is displaced to the right of the first. Four such pairs of rows are produced to form a chessboard. The BBC Micro builds each square from a pair of triangles, a facility not available on the Spectrum.

You can now rewrite the program to RUN on the Spectrum. Because so much of the program is in BBC BASIC, there is nothing to be gained by trying to translate it line by line. It's quicker and easier to examine the program and then work out how the Spectrum could best produce the same result:

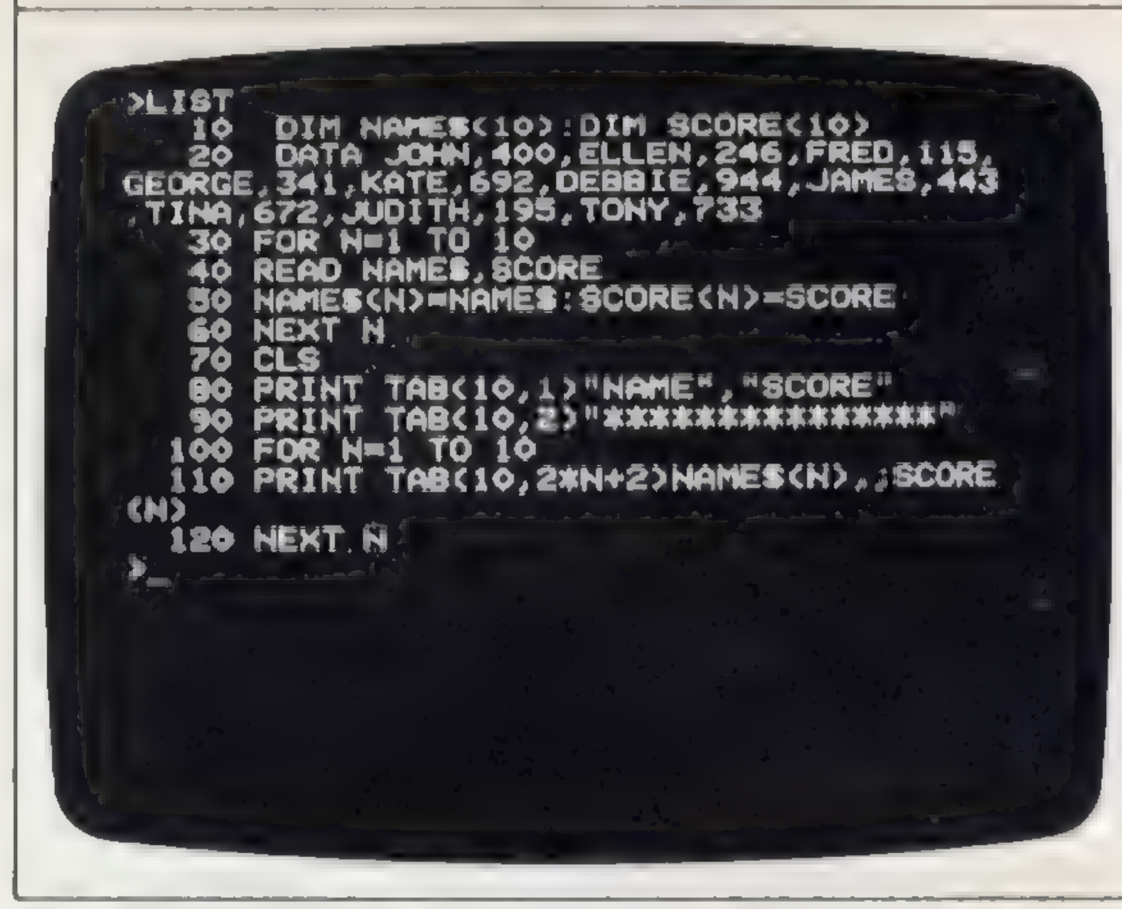
SPECTRUM GAMES BOARD PROGRAM



More BASIC differences

The next program, also written in BBC BASIC, lists ten names on the screen. Each name is accompanied by a number – it might be a list of the best scores for a game. At first sight the method looks quite familiar in that it uses arrays, and should be easily adaptable for the Spectrum:

BBC MICRO ARRAY PROGRAM



The Spectrum can produce this, but watch out for a typical hidden problem. The Spectrum dimensions arrays in a different way. The BBC's name array is dimensioned by:

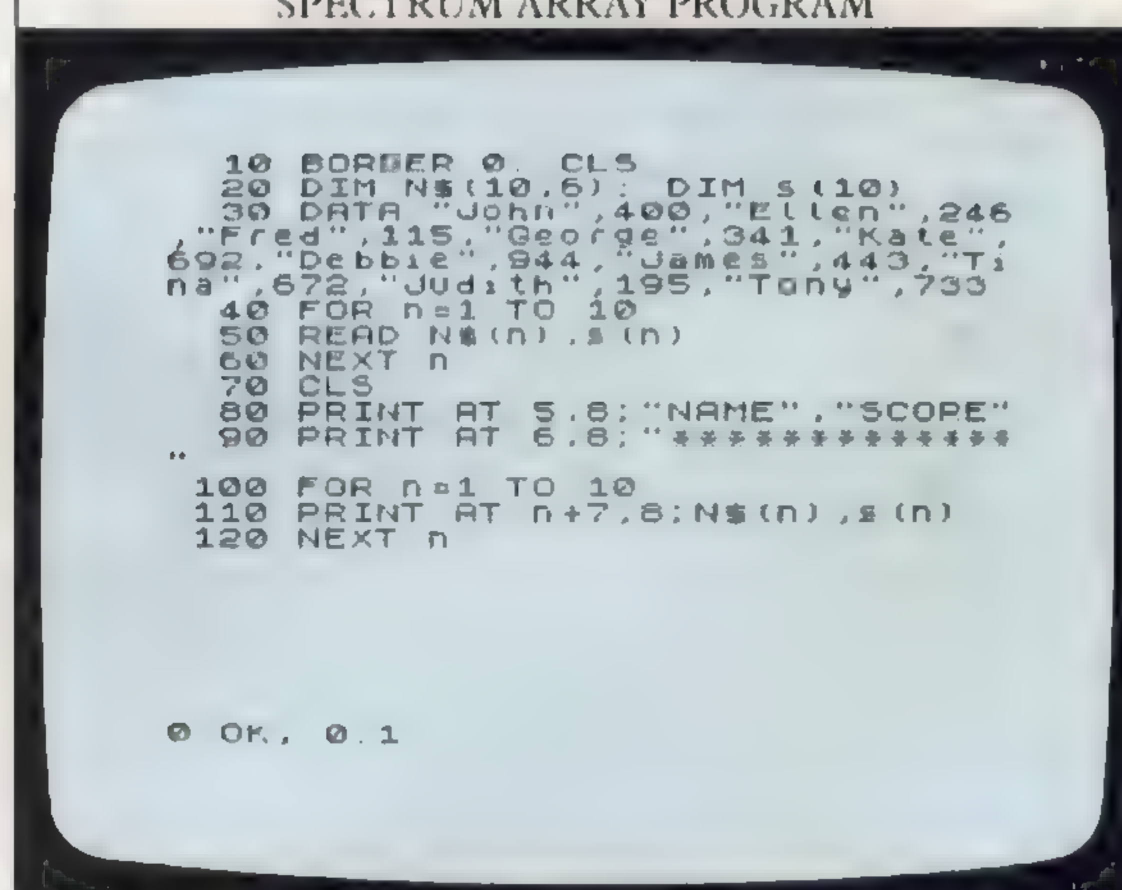
10 DIM NAME$(10)

but the Spectrum requires the length of the strings to be specified as well:

20 DIM N$(10,6)

Also the strings in the Spectrum's DATA must be enclosed in quotation marks. Otherwise the programs are very similar:

SPECTRUM ARRAY PROGRAM



When you are trying to convert programs that seem to use Spectrum-compatible commands, be careful that you do not fall into the trap of assuming that they operate in exactly the same way. Standard words like DIM, INKEY$, TAB and so on, can make converting tricky if you do not realize that many computers use the same commands quite differently.

USING A PRINTER

Although you can write, edit and RUN programs on screen and store them on cassette, a paper print-out is still the best way of examining a listing closely. A print-out can also let you keep the product of a program RUN, which is very useful if you want to look over the results again. So, sooner or later, you will want to add a printer to your system.

The Sinclair printer is of the thermal/electrostatic type. Sparks jump from a pair of moving metal needles onto the special aluminium-coated paper, burning off the aluminium in places to reveal a black ink layer underneath. The spark pattern, and therefore the character printed, is controlled by the computer.

Printer enabling commands

The printer is connected up to the computer by the edge connector on the Spectrum's back panel. When the printer is plugged in, it won't print anything unless you tell it to. PRINT statements in a program send characters to the television screen; the printer ignores them completely.

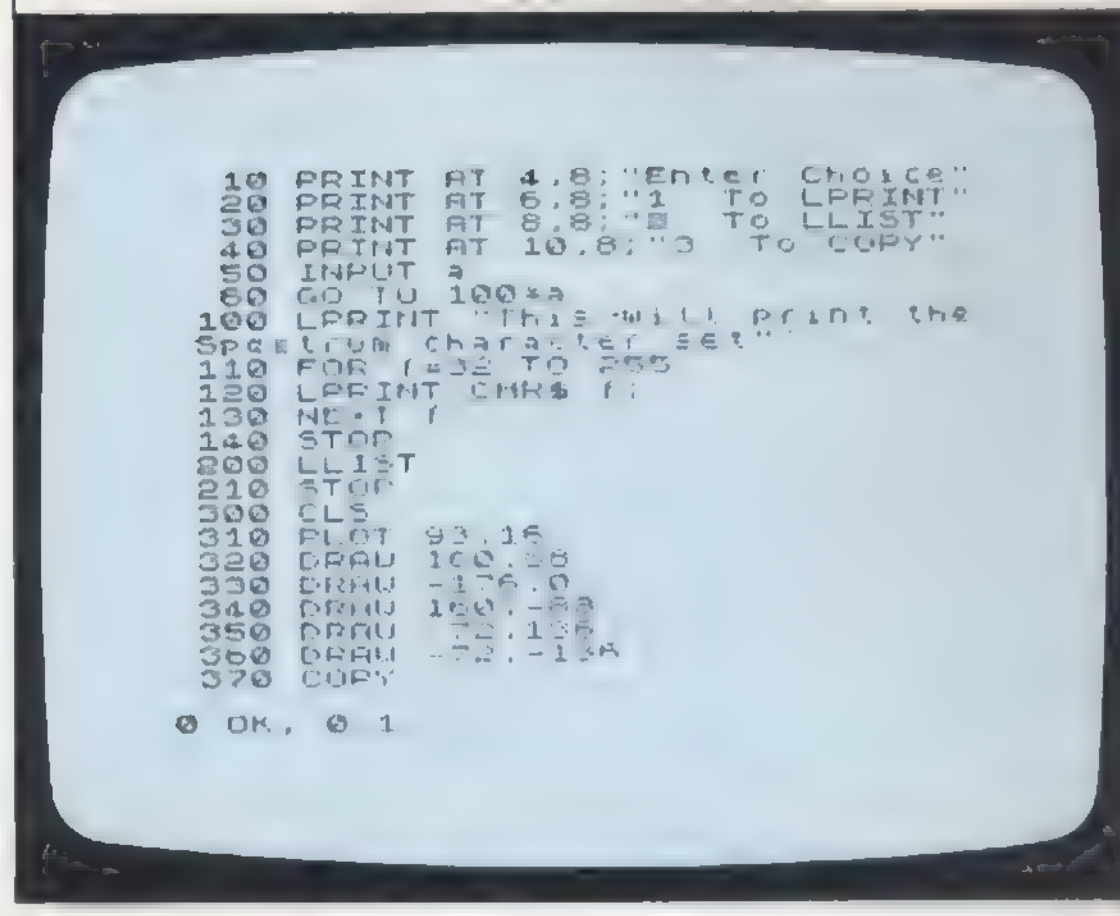
There are several ways of getting information out to the printer. The program opposite enables you to see all of them at work, offering you a choice of the three printer commands. The commands are designed to allow you to use the printer selectively. You can either produce a printed version of what the program will PRINT, or produce a printed listing, or see a copy of the screen display produced on paper.

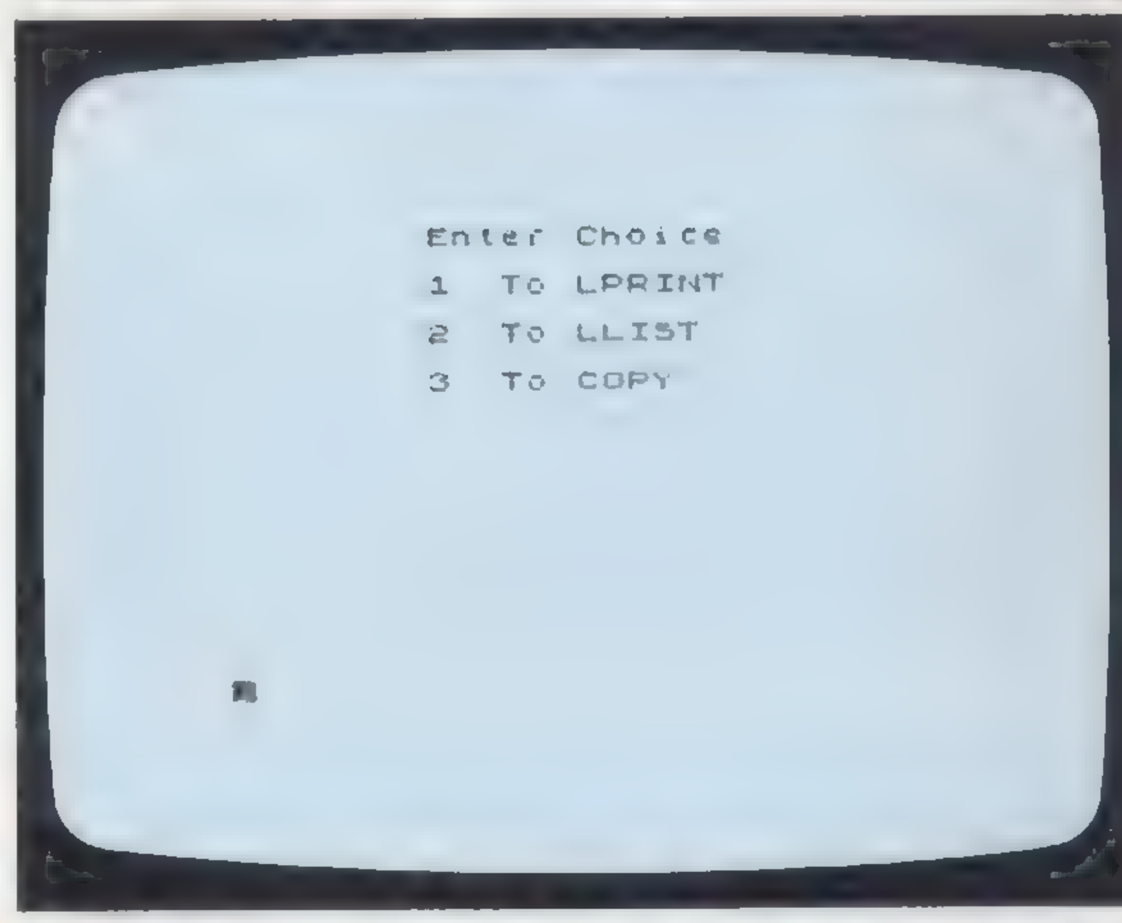
First of all, the printer equivalent of PRINT is LPRINT. It is used just like PRINT but with LPRINT nothing appears on the screen. If you ENTER 1 with the program, the results of a program routine are directed to the printer instead. In this case, the printer will produce a copy of the Spectrum's character set.

The printer's equivalent of LIST is LLIST, selected in the program by ENTERing 2. Again, this only works on the print-out, and will not produce a listing on the screen.

The final method of using the printer is controlled by the COPY command, selected in the program by ENTERing 3. This makes whatever is displayed on the screen appear on the print-out. This is the command used to copy graphics displays, charts and so on.

PRINT-OUT PROGRAM



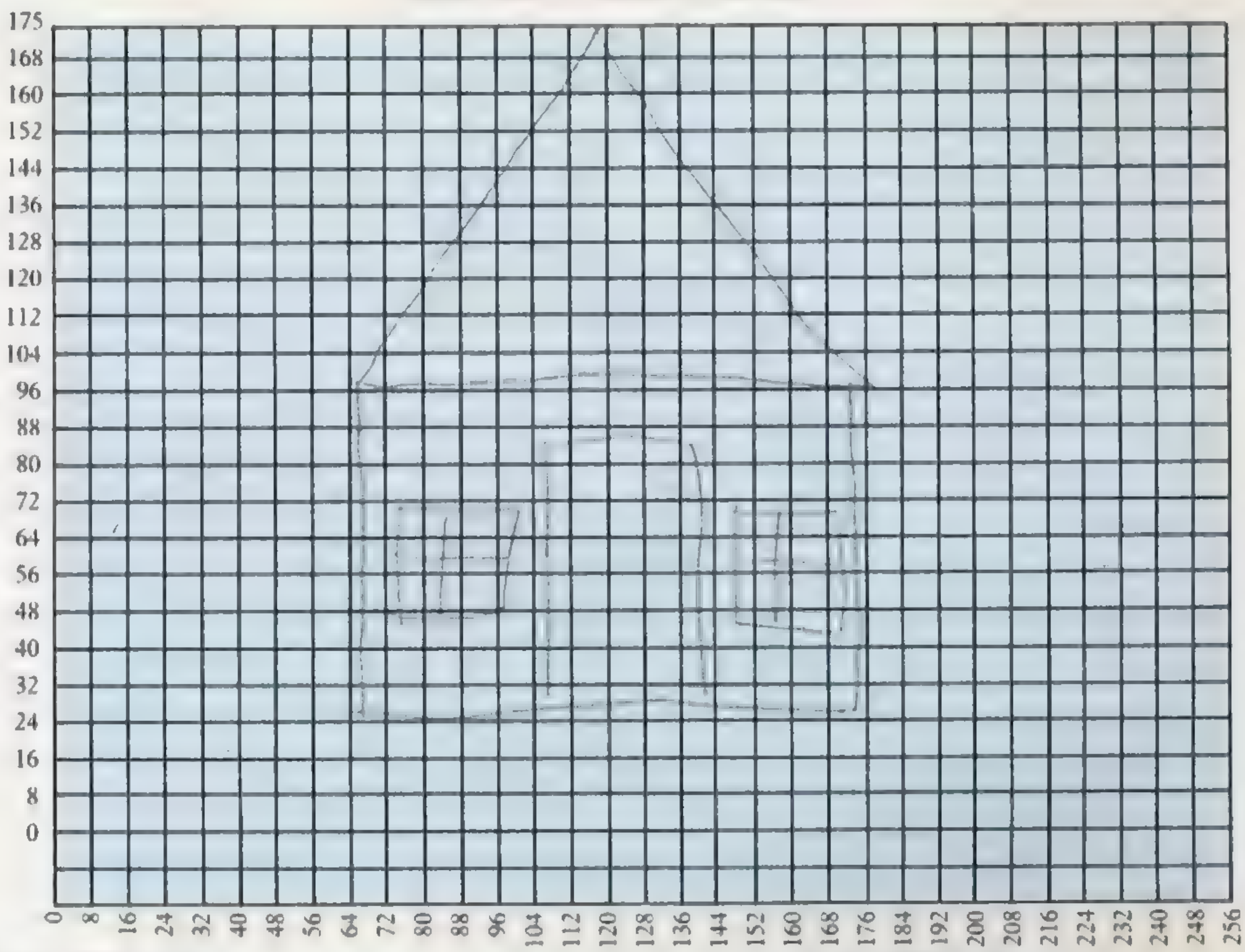


The ZX Printer The printer used with the Spectrum is controlled by three keywords that can be used in programs.

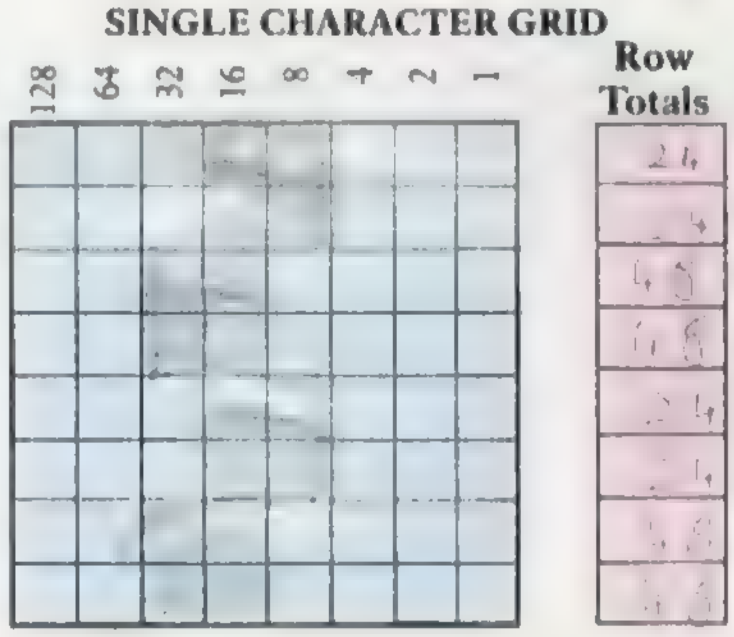
GRAPHICS AND CHARACTER GRIDS

The grid below shows the co-ordinates of the screen display when graphics commands are used. A point on the screen is identified by two co-ordinates x,y. The first co-ordinate sets the horizontal position which is measured along from the left-hand side of the screen. The second co-ordinate sets the vertical position measured from the bottom of the screen. A character PRINTed on the screen occupies an area that is 8 graphics units wide and 8 graphic units high. You cannot PRINT on the bottom two lines of the screen.

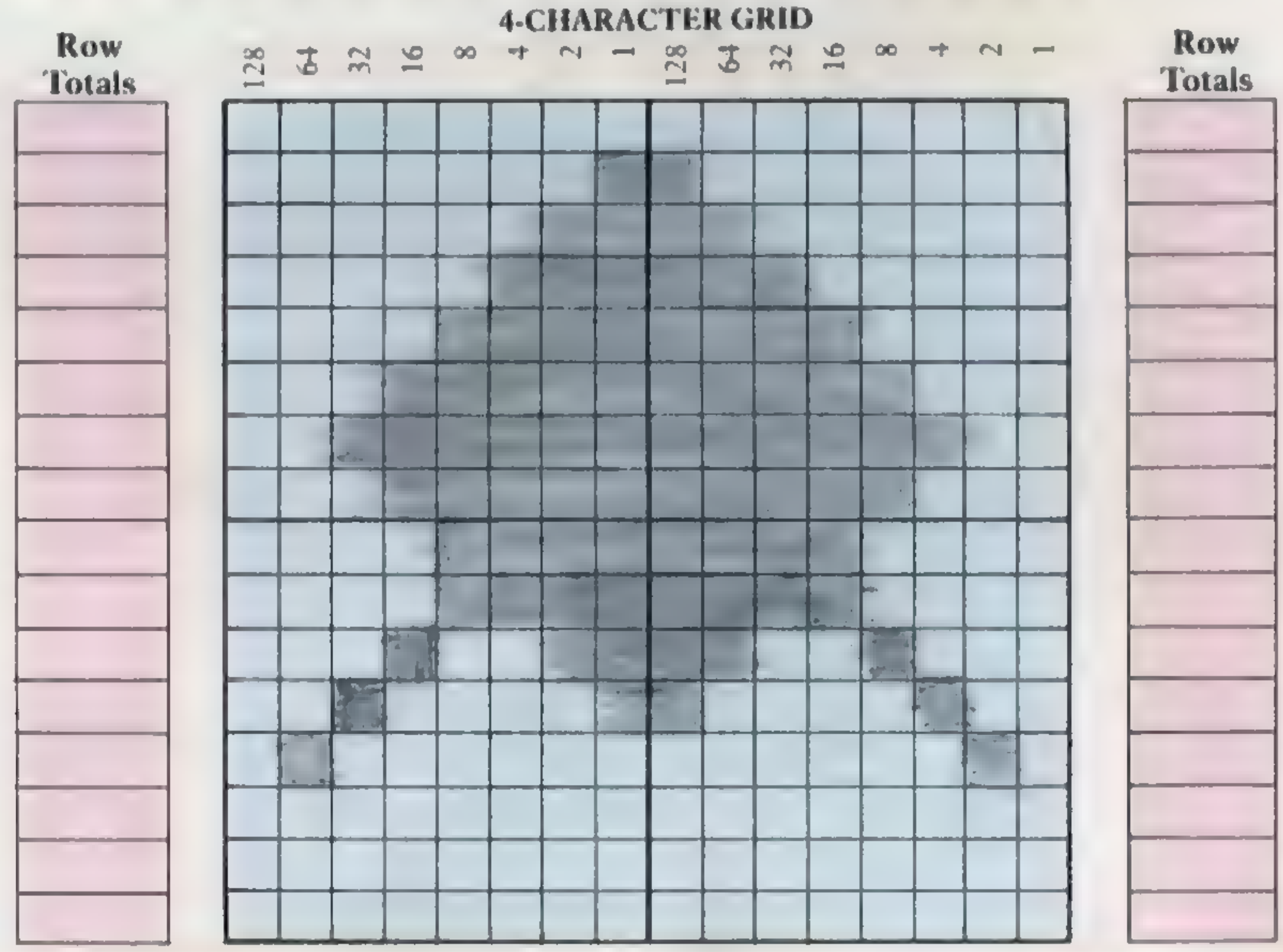
GRAPHICS CO-ORDINATES GRID



SINGLE CHARACTER GRID



4-CHARACTER GRID



Character grids These grids can be used to design either a single character *(above)* or a symbol made from up to four characters *(right)*. You can pencil in your design on the grids and then use the blue columns to list the row totals. These are used in a program with the commands POKE USR. Keys A to U are free for programming user-defined characters.

THE SPECTRUM CHARACTER SET

Each symbol or keyword that the Spectrum uses is represented by a code number. There are 256 code numbers altogether (0–255), each of which can be converted into a single byte of eight binary digits. The letter S for example is specified by CHR$ 83, or CHR$ BIN 1010011. The computer recognizes the binary forms of the code numbers as instructions or information needed to carry out programs. Codes 33 to 126 are allocated to characters specified by the ASCII standard which is used by most microcomputers. The Spectrum's keywords and graphics symbols are specified by the "spare" codes outside this range.

	0	1	2	3	4	5	6	7	8	9
0							PRINT comma	EDIT	cursor left	cursor right
10	cursor down	cursor up	DELETE	ENTER	number		INK control	PAPER control	FLASH control	BRIGHT control
20	INVERSE control	OVER control	AT control	TAB control						
30			space	!	”	#	$	%	&	’
40	(	)	★	+	,	−	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	/	]	↑	—	£	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	-	©		▝
130	▘	▀	▗	▐	▚	▜	▖	▞	▌	▛
140	▄	▟	▙	█	(a)	(b)	(c)	(d)	(e)	(f)
150	(g)	(h)	(i)	(j)	(k)	(l)	(m)	(n)	(o)	(p)
160	(q)	(r)	(s)	(t)	(u)	RND	INKEY$	PI	FN	POINT
170	SCREEN$	ATTR	AT	TAB	VAL$	CODE	VAL	LEN	SIN	COS
180	TAN	ASN	ACS	ATN	LN	EXP	INT	SQR	SGN	ABS
190	PEEK	IN	USR	STR$	CHR$	NOT	BIN	OR	AND	<=
200	>=	<>	LINE	THEN	TO	STEP	DEF FN	CAT	FORMAT	MOVE
210	ERASE	OPEN #	CLOSE #	MERGE	VERIFY	BEEP	CIRCLE	INK	PAPER	FLASH
220	BRIGHT	INVERSE	OVER	OUT	LPRINT	LLIST	STOP	READ	DATA	RESTORE
230	NEW	BORDER	CONTINUE	DIM	REM	FOR	GO TO	GO SUB	INPUT	LOAD
240	LIST	LET	PAUSE	NEXT	POKE	PRINT	PLOT	RUN	SAVE	RANDOMIZE
250	IF	CLS	DRAW	CLEAR	RETURN	COPY				

GLOSSARY

Entries in **bold type** are BASIC keywords.

AND
Allows a program to take a particular course of action only if two conditions are met. An extension of **IF … THEN**.

Array
A collection of **DATA** organized so that each item is labelled and can be handled separately.

AT
Used with **PRINT** to place characters on the screen.

Bar chart
A type of graph where numerical data is represented by columns.

BASIC
Beginners' All-purpose Symbolic Instruction Code; the most commonly used high-level programming language.

BEEP
Makes the computer sound a short tone or beep, whose duration and pitch are determined by the numbers following the command.

BIN
Converts a number written in binary into the equivalent number in decimal.

Binary
A counting system used by computers based on only two numbers – 0 and 1.

Bisection search
A method of searching **DATA** for one particular item. The **DATA** is continually bisected, or divided into two, until the item in question is found.

Bit
A binary digit – 0 or 1.

BORDER
Changes the colour of the screen's border area.

BRIGHT
Turns specified characters to a brighter shade of their **INK** colour.

Byte
A group of eight bits.

Chip
A single package containing a complete electronic circuit. Also called an integrated circuit (IC).

CHR$
Translates the number following from a character code into the equivalent character.

CIRCLE
Draws a circle of a specified size with its centre at a specified point on the screen.

CLS
Clears the text area of the screen.

COPY
Instructs a printer to print out a copy of the screen display.

COS
The trigonometric cosine function.

CPU
Central Processing Unit. Normally contained in a single chip called a microprocessor, this carries out the computer's arithmetic and controls operations in the rest of the computer.

Cursor
A flashing symbol on the screen, showing where the next character will appear.

DATA
The computer treats whatever follows **DATA** as information that may be needed later in the program. Used in conjunction with **READ**.

Debugging
The process of ridding a program of errors or bugs.

DEF FN
Defines a function used elsewhere in a program by the command **FN**.

DIM
Informs the computer about the dimensions of an array so that the computer knows how many items the array contains.

DRAW
Draws a line in the current **INK** colour from the graphics origin at 0,0 or from the last point visited to a point specified.

FLASH
Makes characters flash on the screen.

Flowchart
A diagrammatic representation of the steps necessary to solve a problem.

FN
Indicates that the variable following is being used as the name of a function. The function must be defined by a **DEF FN** statement.

FOR ... NEXT
A loop which repeats a sequence of program statements a specified number of times.

GOSUB
Makes the program jump to a subroutine beginning at the line number following the command. The subroutine must always be terminated by **RETURN**.

GOTO
Makes a program jump to the line number following the command.

Hardware
The physical machinery of a computer system, as distinct from the programs (software) that make it do useful work.

IF ...THEN
Prompts the computer to take a particular course of action only if the condition specified is detected.

INK
Changes the colour of text and graphics that appear on the screen.

INKEY$
Monitors the keyboard to see if any key has been pressed, and if so returns the character.

INPUT
Instructs the computer to wait for some data from the keyboard which is then used in a program.

INT
Converts a number with decimals into a whole number.

Interface
The hardware and software connection between a computer and another piece of equipment.

INVERSE
Switches the **PAPER** colour for the **INK** colour and vice versa.

K
Abbreviation of kilobyte (1024 bytes).

LEN
Counts the total number of characters in a string that follows it.

LET
Assigns a value to a variable.

LIST
Makes the computer display the program currently in its memory.

LLIST
Instructs a printer to print out a listing of the program currently in memory.

LOAD
Transfers a program from a cassette tape into the computer's memory.

Loop
A sequence of program statements which is executed repeatedly or until a specified condition is satisfied.

LPRINT
Sends whatever follows the command to a printer instead of to the screen, so a paper copy only is produced.

MERGE
Allows a second program to be **LOAD**ed into the computer from a tape cassette without erasing the program currently in memory, as long as the line numbers do not coincide.

NEW
Removes a program from the computer's memory.

OR
Allows a program to take a particular course of action if either of two specified conditions are met. An extension of **IF ... THEN**.

OVER
Allows new characters to be **PRINT**ed on top of existing characters.

PAPER
Changes the screen's background colour.

PAUSE
Halts a program for a period set by a number measured in fiftieths of a second.

PEEK
Reports the number stored in a specified location in the memory.

Peripheral
An extra piece of equipment which can be added to the basic computer system – a printer, for example.

Pie chart
A graphic display of numerical data in the form of a divided circle. The size of each slice in the pie reflects the size of the number it represents.

PLOT
Makes a dot appear on the screen at the point specified.

POINT
Reports whether the point at the co-ordinates following is displayed as an **INK** colour or **PAPER** colour.

POKE USR
Stores a number that reprograms a key to produce a user-defined character. On its own, **POKE** puts a number into a specified memory location.

PRINT
Makes whatever follows appear on the screen.

RAM
Random Access Memory (volatile memory). A memory whose contents are erased when the power is switched off. (See also ROM.)

RANDOMIZE
Sets the point at which the fixed **RND** sequence will begin.

READ
Instructs the computer to take information from a **DATA** statement.

REM
Enables the programmer to add remarks to a program. The computer ignores whatever follows the commands.

RESTORE
Resets the point from which **DATA** items are **READ**, so that items can be used more than once in a program.

RETURN
Terminates a subroutine. (See also **GOSUB**.)

RND
Produces numbers between 0 and 1 at random.

ROM
Read Only Memory (non-volatile memory). A memory which is programmed permanently by the manufacturer and whose contents can only be read by the user's computer.

SAVE
Records a program currently in the computer's memory onto a tape. The program is identified by a filename.

SCREEN$
Holds the details of a screen display so that they can be **SAVE**d or **LOAD**ed.

SIN
The trigonometric sine function.

Software
Computer programs.

SQR
Produces the square root of the number that follow:

Statement
An instruction in a program. There may be more than one statement in each program line.

STEP
Sets the step size in a **FOR ... NEXT** loop.

STOP
Halts a program and **PRINT**s out the line number in which it appears.

String
A sequence of characters treated as a single item – someone's name, for instance.

Subroutine
A part of a program that can be called when necessary, to produce a particular display or carry out a number of calculations repeatedly, for example.

Syntax
Rules governing the way statements must be put together in computer language.

TAB
Used with **PRINT** to specify how far along a line characters are to appear.

VAL
Evaluates a number-string, and produces a number.

Variable
A labelled slot in the computer's memory in which information can be stored and retrieved later in a program.

VERIFY
Checks that a program that is currently in memory has been recorded correctly on a tape cassette using **SAVE**.

INDEX

Main entries are in **bold type**

Screen Shot
PROGRAMMING SERIES

The original and exciting new teach-yourself programming course for ZX Spectrum owners.

Over 150 unique screen-shot photographs of program listings *and* programs in action – showing on the page exactly what appears on the screen.

Packed full of programming tips and techniques, reference charts and tables, and advice on how to get the most out of your ZX Spectrum.

CONTENTS INCLUDE

Question-and-answer conversations • Multiple-choice displays • 'Natural' graphics • Sorting and fact-finding • Curves and circles • Pies and slices • Bars and graphs • A guide to writing games • Multiple characters and animation techniques

Further volumes in the Screen Shot **series include**

Step-by-Step Programming for the **ZX Spectrum** BOOK ONE

PLUS

Step-by-Step Programming for the **BBC Micro, Commodore 64, Acorn Electron,** and **Apple II**

DORLING KINDERSLEY

ISBN 0-86318-031-0

00595

£5.95

9 780863 180316